Ingres® 2006 Release 2

Embedded QUEL Companion Guide

INGRES®

February 2007

Contents

Chapter 1: About This Guide

Chapter 2: Embedded QUEL for C

Chapter 3: Embedded QUEL for COBOL

Chapter 4: Embedded QUEL for Fortran

Chapter 5: Embedded QUEL for Ada

Chapter 6: Embedded QUEL for BASIC

Chapter 7: Embedded QUEL for Pascal

Index

Chapter 1: About This Guide

Overview

This chapter briefly describes the *Embedded QUEL Companion Guide* and discusses how to use this manual most effectively. The chapter also describes conventions used in this documentation and lists other manuals that are relevant to this manual.

Purpose of This Manual

This guide describes how to use Embedded QUEL (EQUEL) with the following programming languages:

- C and C++
- COBOL
- Fortran
- Ada
- BASIC
- Pascal

For the most part EQUEL is identical in syntax and functionality across all supported host programming languages. Therefore the documentation describes it independently of any one host language in the *QUEL Reference Guide*, which covers database statements, and in the *Forms-based Application Development Tools User Guide*, which covers forms statements. The host language-dependent details of its use are described in this Companion Guide.

Audience

This manual is designed for programmers who have a working knowledge of QUEL and C, COBOL, Fortran, Ada, BASIC, or Pascal. It must be read in conjunction with the *QUEL Reference Guide* and the *Forms-based Application Development Tools User Guide*, as it discusses only those issues on which the various host languages diverge.

Contents

Each chapter in this guide discusses EQUEL for a particular host language. Each chapter contains the following sections:

Section	Description
EQUEL Statement Syntax	Language-specific issues of EQUEL statement syntax
Variables and Data Types	Declaration and use of language-specific program variables in EQUEL
Dynamically Built Param Statements	The **param** feature that dynamically builds EQUEL statements. **Note:** This feature is supported in EQUEL/C and EQUEL/Fortran only.
Runtime Error Processing	A user-defined EQUEL error handler
Precompiling, Compiling and Linking an EQUEL Program	The EQUEL preprocessor for the host language and the steps required to create, compile, and link an EQUEL program
Preprocessor Error Messages	EQUEL preprocessor error messages specific to the host language
Remaining sections	Sample programs that illustrate many EQUEL features

Conventions

This section describes the conventions that Ingres documentation uses for consistency and clarity.

Statements and Commands

Ingres documentation handles statements and commands as follows.

Terminology

The documentation observes the following distinction in terminology:

- A *command* is an operation that you execute at the operating system level
- A *statement* is an operation that you embed within a program or execute interactively from the Terminal Monitor

A statement can be written in 4GL, a host programming language (such as C), or a database query language (SQL or QUEL).

Syntax

This manual uses the following conventions to describe statement and command syntax specifications:

Convention	Usage
Boldface	Indicates keywords, symbols or punctuation that you must type as shown
Italic	Represents a variable name for which you must supply an actual value
[ ] (brackets)	Indicate an optional item
{ } (braces)	Indicate an optional item that you can repeat as many times as appropriate
| (vertical bar)	Used between items in a list to indicate that you should choose one of the items

The following example illustrates the syntax conventions:

create table *tablename* **(***columnname format*
*{,columnname format}***)**
[**with**_*clause*]

System Specific Text

Although Ingres generally operates the same way on all systems, there are a few system-specific differences you need to know about. Where information differs by system, read the information that follows the name of your system, as follows:

UNIX This text is specific to the UNIX environment.

VMS This text is specific to the VMS environment.

Windows This text is specific to the Windows environment.

The symbol indicates the end of the system-specific text.

In some instances, system-specific differences are indicated by using parenthesis ( ). For example: This is useful for program libraries that are using **make** dependencies (UNIX) or MMS dependencies (VMS).

Related Manuals

This guide is part of a series of manuals that describe the full range of Ingres products.

To learn more about concepts and functions related to EQUEL, see the following manuals:

- *QUEL Reference Guide*
- *Character-based Querying and Reporting Tools User Guide*
- *Forms-based Application Development Tools User Guide*

Chapter 2: Embedded QUEL for C

This chapter describes the use of EQUEL with the C and C++ programming languages.

EQUEL Statement Syntax for C

This section describes the language-specific ground rules for embedding QUEL database and forms statements in a C or C++ program. An EQUEL statement has the following general syntax:

*EQUEL_statement*

For information on QUEL statements, see the *QUEL Reference Guide*. For information on EQUEL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe how to use the various syntactical elements of EQUEL statements as implemented in C.

Margin

There are no specified margins for EQUEL statements in C. Always place two number signs (**##**) in the first two positions of a line. The rest of the statement can begin anywhere else on the line.

Terminator

An EQUEL/C statement does not need a statement terminator. It is conventional not to use a statement terminator in EQUEL statements. However, you can use the C statement terminator, the semicolon (;), at the end of EQUEL statements because the preprocessor ignores it.

For example, the preprocessor considers the following two statements as the same:

```
##  sleep 1
```

and

```
##  sleep 1;
```

EQUEL statements that are made up of a few other statements, such as a **display** loop, only allow a semicolon after the last statement. For example:

```
##   display empform            /* No semicolon here */
##   initialize                 /* No semicolon here */
##   activate menuitem "Help"   /* No semicolon here */
##   {
##     message "No help yet";    /* Semicolon allowed */
##     sleep 2;                  /* Semicolon allowed */
##   }
##   finalize;  /* Semicolon allowed on last statement */
```

When using a **retrieve** loop, place a semicolon after the retrieve statement to disassociate the loop code inside the braces from the **retrieve** statement itself. Variable declarations made visible to EQUEL follow the normal C declaration syntax. Thus, you must terminate variable declarations in the normal way for C, with a semicolon.

Line Continuation

There are no special line-continuation rules for EQUEL/C. You can break an EQUEL statement between words and continue it on any number of subsequent lines. An exception to this rule is that you cannot continue a statement between two words that are reserved when they appear together, such as **declare cursor**. For a list of double keywords, see the *QUEL Reference Guide*. Start each continuation line with ## characters. You can put blank lines between continuation lines.

If you want to continue a character-string constant across two lines, end the first line with a backslash character (\), and continue the string at the beginning of the next line. In this case, do not place ## characters at the beginning of the continuation lines.

For examples of string continuation, see String Literals in this chapter.

Comments

Two kinds of comments can appear in an EQUEL program: EQUEL comments and host language comments. The /* and */ characters delimit EQUEL comments and must appear on lines beginning with the ## sign.

Because the EQUEL comment delimiters are the same as those for the C language, all comments appearing on EQUEL lines in a C program (those beginning with ##) are treated as EQUEL comments. Whereas the preprocessor passes C comments through as part of its output, it strips EQUEL comments out of the program and does not pass them through. Thus, source code comments that you desire in the preprocessor output should be entered as C comments—on lines other than EQUEL lines.

The following restrictions apply to any EQUEL or C comments in an EQUEL/C program:

- If anything other than ## appears in the first two positions of a line of EQUEL source, the precompiler treats the line as host code and ignores it. The only exception to this is a string-continuation line. For details, see String Literals in this chapter.
- Comments cannot appear in string constants. If this occurs, the preprocessor interprets the intended comment as part of the string constant.
- In general, you can put EQUEL comments in EQUEL statements wherever you can legally put a space. However, comments cannot appear between two words that are reserved when they appear together, such as **declare cursor**. See the list of EQUEL reserved words in the *QUEL Reference Guide.*

The following additional restrictions apply only to C comments in an EQUEL/C program:

- C comments cannot appear between component lines of EQUEL block-type statements. These include **retrieve**, **initialize**, **activate**, **unloadtable**, **formdata**, and **tabledata**, all of which have optional accompanying blocks delimited by open and close braces. C comment lines cannot appear between the statement and its block-opening delimiter.

  For example:

```
## retrieve (ename = employee.name)
      /* Illegal to put a host comment here! */
## {
      /* A host comment is legal here */
      printf ("Employee name: %s", ename);
## }
```

- C comments cannot appear between the components of compound statements. In particular, it is illegal for a C comment to appear between any two adjacent components of the **display** statement. This includes **display** itself and its accompanying **initialize**, **activate**, and **finalize** statements.

  For example:

```
## display empform
      /* Illegal to put a host comment here! */
## initialize (empname = "Fred McMullen")
      /* Host comment illegal here! */
## activate menuitem "Clear":
## {
/* Host comment here is fine */
## clear field all
## }
      /* Host comment illegal here! */
## activate menuitem "End":
## {
##      breakdisplay
## }
      /* Host comment illegal here! */
## finalize
```

The *QUEL Reference Guide* specifies these restrictions on a statement-by-statement basis.

EQUEL comments are legal, however, in the locations the previous paragraph describes, as well as wherever a host comment is legal. For example:

```
## retrieve (ename = employee.name)
## /*   This is an EQUEL comment, legal in this
##      location and it can span multiple lines */
## {
      printf ("Employee name %s", ename);
## }
```

String Literals

You can use either double quotes (") or single quotes (') to delimit string literals in EQUEL/C. Be sure that you begin and end each string literal with the same delimiter.

Whichever quote mark you use, you can embed it as part of the literal itself. Just precede it with a backslash. For example:

```
##   append comments
##   (field1 = "a double \" quote is in this string")
```

or

```
##   append comments
##   (field1 = 'a single \' quote is in this string')
```

To include the backslash character as part of the string, precede it with another backslash.

To continue a string literal to additional lines, use the backslash character (\). The preprocessor ignores the backslash and the following newline character, so that the following line can continue both the string and any further components of the EQUEL statement. Any leading spaces on the next line are considered part of the string. This follows the C convention. For example, the following EQUEL statements are legal:

```
##   message 'Please correct errors found in updating \
the database tables.'

##   append to employee (empname = "Freddie \
Mac", empnum = 222)
```

C Variables and Data Types

This section describes how to declare and use C program variables in EQUEL.

Variable and Type Declarations

The following section describes variable and type declarations.

EQUEL Variable Declaration Procedures

You must make known to the processor any C language variable that you use in an EQUEL statement so that it can determine the type of the variable. You must precede the variable declaration in an EQUEL/C program by two number signs (**##**) that begin in the first column position of the line. If a variable is not used in an EQUEL statement, you do not need to use number signs.

Reserved Words in Declarations

In declarations, all EQUEL keywords are reserved. Therefore, you cannot declare types or variables with the same name as those keywords. Also, the following EQUEL/C keywords, used in declarations, are reserved and cannot be used elsewhere, except in quoted string constants:

auto	**extern**	**long**	**typedef**
char	**float**	**register**	**union**
define	**globaldef**	**short**	**unsigned**
double	**globalref**	**static**	**varchar**
enum	**int**	**struct**	

Note that not all C compilers reserve every keyword listed. However, the EQUEL/C preprocessor does reserve all these words.

The EQUEL preprocessor does not distinguish between uppercase and lowercase in keywords. When generating C code, it converts any uppercase letters in keywords to lowercase. The following example shows that although the following declarations are initially unacceptable to the C compiler, the preprocessor converts them into legitimate C code. Lines without **##** in the first two column positions pass through without case conversion.

```
## defINE ARRSIZE 256
##      /* "defINE" is converted to "define" */
## INT numarr[ARRSIZE];
##      /* "INT" is equivalent to "int" */
```

The rule just described is true only for keywords. The preprocessor does distinguish between case in program-defined types and variable names.

Variable and type names must be legal C identifiers beginning with an underscore or alphabetic character.

Data Types

The EQUEL/C preprocessor accepts the C data types in the following table and maps them to corresponding Ingres types. For further information on type mapping between Ingres and C data, see Data Type Conversion in this chapter.

C Data Types and Corresponding Ingres Types

C Data Type	Ingres Data Type
long	integer
int	integer
short	integer
char (no indirection)	integer
double	float
float	float
char * (character pointer)	character
char [] (character array)	character
unsigned	integer
unsigned int	integer
unsigned long	integer
unsigned short	integer
unsigned char	integer
long int	integer
short int	integer
long float	float

The Integer Data Type

The EQUEL preprocessor accepts all C integer data types. Even though some integer types do have C restrictions (for example, values of type **short** must be in the limits of your machine). The preprocessor does not check these restrictions. At runtime, data type conversion is determined according to standard C numeric conversion rules. For details on numeric type conversion, see Data Type Conversion in this chapter.

The type adjectives **long**, **short**, or **unsigned** can qualify the integer type.

In the type mappings table just presented, the C data type **char** has three possible interpretations, one of which is the Ingres **integer** data type. The adjective **unsigned** can qualify the **char** type when it is used as a single-byte integer. If a variable of the **char** data type is declared without any C indirection, such as an array subscript or a pointer operator (the asterisk), it is considered a single-byte integer variable. For example:

```
##  char        age;
```

is a legal declaration and can be used to hold an integer Ingres value. If the variable is declared with indirection, it is considered an Ingres character string.

You can use an integer variable with any numeric-valued object to assign or receive numeric data. For example, you can use it to set a field in a form or to select a column from a database table. It can also specify simple numeric objects, such as table field row numbers.

The following example shows the way several C data types are interpreted by EQUEL:

```
## char  age;              /* Single-byte  integer  */
## short empnums[MAXNUMS];  /* Short  integers  array*/
## long  *global_index;  /* Pointer  to  long  integer  */
## unsigned  int    overtime;
```

The Floating-point Data Type

The preprocessor accepts **float** and **double** as floating-point data types. The internal format of **double** variables is the standard C runtime format.

You can only use a floating-point variable to assign or receive numeric data. You cannot use it to specify numeric objects, such as table field row numbers. Note that **long float**, a construct allowed in some compilers, is accepted as a synonym for **double**.

```
##  float       salary;
##  double      sales;
```

VMS

If you declare long floating variables to be used with EQUEL statements, you should not compile your program with the **g_float** command line qualifier when you are using the VAX C compiler. This qualifier changes the long float internal storage format, causing runtime numeric errors.

The Character String Data Type

Any variables built up from the **char** data type, except simple variables declared without any C indirection, are compatible with any Ingres character string objects. As previously mentioned, a variable of type **char** declared without any C indirection is considered an integer variable. The preprocessor treats an array of characters and a pointer to a character string in the same way. Always null-terminate a character string if the string is to be assigned to an Ingres object. Ingres automatically null terminates any character string values that are retrieved into C character string variables. Consequently, any variable that you use to receive Ingres values must be declared as the maximum object length, plus one extra byte for the C null string terminator. For more information, see Runtime Character Conversion in this chapter.

The following example declares three character variables—one integer and two strings:

```
## char    age;    /* Single byte integer */
## char    *name;  /* To be a pointer to a string */
## charbuf[16];
/*
** To be used to receive at most 15 bytes of string
** data, plus a null string terminator
*/
```

For more information on character strings that contain embedded nulls, see The Varying Length String Type in this chapter.

Define Declaration

The EQUEL preprocessor accepts the **## define** directive, which defines a name to be a constant value. The EQUEL preprocessor replaces the **## define** statement with the C **# define** statement.

The syntax for the **## define** statement is:

define *constant_name constant_value*

Syntax Notes:

- The *constant_value* must be an integer, floating-point, or character string literal. It cannot be an expression or another name. It cannot be left blank, as would happen if you intend to use it later with the **# ifdef** statement. If the value is a character string constant, you must use double quotes to delimit it. Do not use single quotes to delimit *constant_name* in order to make it be interpreted as a single character constant, because the preprocessor translates the single quotes into double quotes. For example, both of the following names are interpreted as string constants, even though the first may be intended as a character constant:

```
## define QUITFLAG 'Q'
## define ERRORMSG "Fatal error occurred."
```

- The preprocessor does not accept casts before *constant_value*. In general, the preprocessor does not accept casts, and it interprets data types from the literal value.

You can only use a defined constant to assign values to Ingres objects. Attempting to retrieve Ingres values into a constant causes a preprocessor error.

```
## define minempnum 1
## define maxsalary 150000.00
## define defaultnm "No-name"
```

EQUEL does not recognize a C define declaration with only one #.

Variable Declarations Syntax

The syntax of a variable declaration is:

[*storage_class*] *type_specification*
declarator {**,** *declarator*}**;**

where each *declarator* is:

variable_name [**=** *initial_value*]

Syntax Notes:

- *Storage_class* is optional, but if specified can be any of the following:

  auto
  extern
  register
  static
  varchar

VMS also uses **globaldef** and **globalref** unless you are using ANSI C on VMS.

The storage class provides no data type information to the preprocessor. For more detail on the EQUEL-defined **varchar** storage class, see The Varying Length String Type in this chapter.

- Although register variables are supported, be careful when using them in EQUEL statements. In database statements, such as the **append** and **retrieve** statements, the preprocessor generates C function calls which may pass a variable by reference using the ampersand operator (&). However, some compilers do not allow you to use register variables in this manner.
- Because of the syntactic similarity between the EQUEL **register** statement and the C **register** declaration, the preprocessor does not allow you to represent the initial object name in the EQUEL **register** statement with a host variable.
- The *type_specification* must be an EQUEL type, a type built up with a **typedef** declaration (and known to the preprocessor), or a structure or union specification. For a discussion of **Typedef** declarations, see Type Declarations Syntax in this chapter. For a discussion of structures, see Structure Declarations Syntax in this chapter.
- Precede the *variable_name* by an asterisk (*), to denote a pointer variable, or follow it by a bracketed expression (**[***expr***]**), to denote an array variable. For a discussion of pointers, see Pointer Declarations Syntax in this chapter. For a discussion of arrays, see Array Declarations Syntax in this chapter.
- Begin the *variable_name* with a legal C identifier name that starts with an underscore or alphabetic character.
- The preprocessor does not evaluate the *initial_value*. Consequently, the preprocessor accepts any initial value, even if it can later cause a C compiler error. For example, the preprocessor accepts both of the following initializations, even though only the first is a legal C statement:

```
## char     *msg = "Try again";
## int      rowcount = {0, 123};
```

The following example illustrates some valid EQUEL/C declarations:

```
## extern  int   first_employee;
## auto    long  update_mode = 1;
## static  char  *names[3] = {"neil", "mark", "barbara"};
## static  char  *names[3] = {"john", "bob", "tom"};
## char          **nameptr = names;
## short         name_counter;
## float         last_salary = 0.0, cur_salary = 0.0;
## double        stat_matrix[STAT_ROWS][STAT_COLS];
```

Type Declarations Syntax

The syntax of a type declaration is:

typedef *type_specification*
typedef_name {**,** *typedef_name*}**;**

Syntax Notes:

- The **typedef** keyword acts like a storage class specifier in a variable declaration, except that the resulting *typedef_name* is marked as a type name and not as a variable name.
- The *type_specification* must be an EQUEL/ C type, a type built up with a **typedef** declaration and known to the preprocessor, or a structure or union specification. For a discussion of structures, see Structure Declarations Syntax in this chapter.
- Precede the *typedef_name* by an asterisk (*), to denote a *pointer type,* or follow it by a bracketed expression ([*expr*]), to denote an array type. For a discussion of pointers, see Pointer Declarations Syntax in this chapter. For a discussion of arrays, see Array Declarations Syntax in this chapter.
- The preprocessor accepts an initial value after *typedef_name*, although you should avoid putting one there because it would not signify anything. Most C compilers allow an initial value that is ignored after the *typedef_name*. The initial value is not assigned to any variables declared with that *typedef*.

```
## typedef       short      INTEGER2;
## typedef       char      CHAR_BUF[2001], *CHAR_PTR;

## INTEGER2      i2;
## CHAR_BUF      logbuf;
## CHAR_PTR      name_ptr = (char *)0;
```

Array Declarations Syntax

The syntax of a C array declaration is:

*array_name***[***dimension***]** {**[***dimension***]**}

In the context of a simple variable declaration, the syntax is:

*type_specification array_variable_name***[***dimension***]** {**[***dimension***]**}**;**

In the context of a type declaration, the syntax is:

typedef *type_specification array_type_name***[***dimension***]**
{**[***dimension***]**}**;**

Syntax Notes:

- The preprocessor does not evaluate the *dimension* specified in the brackets. Consequently, the preprocessor accepts any dimensions, including illegal dimensions such as non-numeric expressions, which later cause C compiler errors.

  For example, the preprocessor accepts both of the following declarations, even though only the second is legal C:

```
## typedef int SQUARE["bad expression"];
/* Non-constant expression */
## int          cube_5[5][5][5];
```

- You can specify any number of dimensions. The preprocessor notes the number of dimensions when the variable or type is declared. When the variable is later referenced, it must have the correct number of indices.
- An array variable can be initialized, but the preprocessor does not verify that the initial value is an array aggregate.
- An array of characters is considered to be the pseudo character string type.

The following example illustrates the use of array declarations:

```
## define COLS 5
## typedef short          SQUARE[COLS][COLS];
## SQUARE                 sq;

## static int             matrix[3][3] =
##                              { {11, 12, 13},
##                              {21, 22, 23},
##                              {31, 32, 33} };

## char                   buf[50];
```

Pointer Declarations Syntax

The syntax of a C pointer declaration is:

*****{*****} *pointer_name*

In the context of a simple variable declaration, the syntax is:

type_specification *****{*****} *pointer_variable_name***;**

In the context of a type declaration, the syntax is:

typedef *type_specification* *****{*****} *pointer_type_name***;**

Syntax Notes:

- You can specify any number of asterisks. The preprocessor notes the number specified when the variable or type is declared. When the variable is later referenced, it must have the correct number of asterisks.
- A pointer variable can be initialized, but the preprocessor does not verify that the initial value is an address.
- A pointer to the **char** data type is considered to be the pseudo character string type.
- You can use arrays of pointers.

The following example illustrates the use of pointer declarations:

```
## extern int        min_value;
## int               *valptr = &min_value;
## char              *tablename = "employee";
```

Structure Declarations Syntax

A C structure declaration has three variants depending on whether it has a tag and/or a body. The following sections describe these variants.

A Structure with a Tag and a Body

The syntax of a C structure declaration with a tag and a body is:

struct *tag_name* **{**
structure_declaration {*structure_declaration*}
};

where *structure_declaration* is:

type_specification
member {**,** *member*}**;**

In the context of a simple variable declaration, the syntax is:

struct *tag_name* **{**
structure_declaration {*structure_declaration*}
} [*structure_variable_name*]**;**

In the context of a type declaration, the syntax is:

typedef struct *tag_name* **{**
structure_declaration {*structure_declaration*}
} *structure_type_name***;**

Syntax Notes:

- Wherever the keyword **struct** appears, the keyword **union** can appear instead. The preprocessor treats them as equivalent.
- Each member in a *structure_declaration* has the same rules as a variable of its type. For example, as with variable declarations, the *type_specification* of each member must be a previously defined type or another structure. Also, you can precede the member name by asterisks or follow it by brackets. Because of the similarity between structure members and variables, the following discussion focuses only on those areas in which they differ.
- A structure member can be a nested structure declaration. For example:

```
## struct person
## {
##        charname[40];
##        struct
##        {
##            int day, month, year;
##        } birth_date;
## } owner;
```

- Only structure members that will be referenced in EQUEL statements need to be declared to EQUEL. The following example declares a C structure with the fileloc member that is not known to EQUEL:

```
##  struct address {
##    int        number;
##    char        street[30];
##    char        town[20];
##    short        zip;
      FILE        *fileloc;    /* Unknown to EQUEL */
## } addr[20];
```

- Although the preprocessor permits an initial value after each member name, do not put one there because it causes a compiler syntax error.
- If you do not specify the *structure_variable_name*, the declaration is considered a declaration of a structure tag.
- A structure variable can be initialized, but the preprocessor does not verify that the initial value is a structure aggregate.

The following example illustrates the use of a tag and a body:

```
## define max_employees 1500
## typedef struct employee
## {
##       char        name[21];
##       short        age;
##       double        salary;
## } employee_desc;
## employee_desc employees[MAX_EMPLOYEES];
## employee_desc *empdex = &employees[0];
```

A Structure with a Body and No Tag

The syntax of a C structure declaration with a body and no tag is:

struct {
structure_declaration {*structure_declaration*}
} *structure_variable_name*;

where *structure_declaration* is the same as in the previous section.

In the context of a simple variable declaration, the structure's syntax is:

struct {
structure_declaration {*structure_declaration*}
} *structure_variable_name***;**

In the context of a type declaration, the structure's syntax is:

typedef struct {
structure_declaration {*structure_declaration*}
} *structure_type_name***;**

Syntax Notes:

- All common clauses have the same rules as in the previous section. For example, **struct** and **union** are treated as equivalent, and the same rules apply to each structure member as to variables of the same type.
- Specify the *structure_variable_name* when there is no tag. In fact, the actual structure definition applies only to the variable being declared.

The following example illustrates the use of a body with no tag:

```
##  define MAX_EMPLOYEES 1500
##  struct
##  {
##       char      name[21];
##       short     age;
##       double    salary;
##  } employees[MAX_EMPLOYEES];
```

A Structure with a Tag and No Body

The syntax of a C structure declaration with a tag and no body is:

struct *tag_name*

In the context of a simple variable declaration, the syntax is:

struct *tag_name structure_variable_name***;**

In the context of a type declaration, the syntax is:

typedef struct *tag_name structure_type_name***;**

Syntax Notes:

- All common clauses have the same rules as in the previous section. For example, **struct** and **union** are treated as equivalent, and the structure can be initialized without the preprocessor checking for a structure aggregate.
- The *tag_name* must refer to a previously defined structure or union. The preprocessor does not support *forward* structure declarations. Therefore, when referencing a structure tag in this type of declaration, the tag must have already been defined. In the declaration below, the tag "new_struct" must have been previously declared:

```
## typedef struct new_struct *NEW_TYPE;
```

The following example illustrates the use of a tag and no body:

```
  ## union a_name
  ## {
  ##     char        nm_full[30];
  ##     struct
  ##     {
  ##     char nm_first[10];
  ##     char nm_mid[2];
  ##     char nm_last[18];
  ##     } nm_parts;
  ## };

## union a_name empnames[MAX_EMPLOYEES];
```

Enumerated Integer Types

An enumerated type declaration, **enum**, is treated as an integer declaration. The syntax of an enumerated type declaration is:

enum [*enum_tag*]
  { *enumerator* [**=** *integer_literal*]
    {**,** *enumerator* [**=** *integer_literal*]} **}** [*enum_vars*];

The outermost braces (**{** and **}**) represent actual braces you type.

Syntax Notes:

- If you use the *enum_tag*, the list of enumerated literals (*enumerators*) and enum variables (*enum_vars*) is optional, as in a structure declaration. The two declarations that follow are equivalent. The first declaration declares an *enum_tag*, while the second declaration uses that tag to declare a variable.

First declaration:

```
## enum color {RED, WHITE, BLUE};
/* Tag, no variable */
## enum color col;    /* Tag, no body, has variable */
```

Second declaration:

```
## enum color {RED, WHITE, BLUE} col;
/* Tag, body, has variable */
```

If you do not use the *enum_tag*, the declaration must include a list of enumerators, as in a structure declaration.

- You can use the **enum** declaration with any other variable declaration, type declaration, or storage class. For example, the following declarations are all legal:

```
## typedef enum {dbTABLE, dbCOLUMN, dbROW,
## dbVIEW, dbGRANT} dbOBJ;

## dbOBJ  obj, objs[10];

## extern  dbOBJ   *obj_ptr;
```

- Enumerated variables are treated as integer variables and enumerated literals are treated as integer constants.

The Varying Length String Type

All C character strings are *null-terminated*. (For more information, see The Character String Data Type in this chapter). Ingres data of type **char** or **varchar** can contain random binary data including the zero-valued byte (the null byte or "\0" in C terms). If a program uses a C **char** variable to retrieve or set binary data that includes nulls, the EQUEL runtime system is not able to differentiate between embedded nulls and the null terminator. Unlike other programming languages, C does *not* blank-pad fixed length character strings.

In order to set and retrieve binary data that may include nulls, a new EQUEL/C storage class, **varchar**, has been provided for varying length string variables. **varchar** identifies the following variable declaration as a structure that describes a varying length string, namely, a 2-byte integer representing the count and a fixed length character array. Like other storage classes previously described, the keyword **varchar** must appear before the variable declaration:

```
## varchar struct {
##        short     current_length;
##        char      data_buffer[MAX_LENGTH];
## } varchar_structure;
```

Syntax Notes:

- The word **varchar** is reserved and can be in uppercase or lowercase.
- The **varchar** keyword is not generated to the output C file.

- The **varchar** storage class can only refer to a variable declaration, not to a type definition. For example, the following declaration is legal because it declares the variable "vch":

```
## varchar struct {
##        short       buf_size;
##        char        buf[100];
## } vch;
```

  But the **varchar** declaration of the structure tag "vch" (without a variable) is *not* legal and will generate an error:

```
## varchar    struct   vch {
##            short     buf_size;
##            char      buf[100];
## };
```

- The structure definition of a **varchar** variable declaration can be replaced by a structure tag or typedef reference. For example the following **typedef** and **varchar** declarations are legal:

```
## typedef    struct   vch_ {
##            short     vch_count;
##            char      vch_data[VCH_MAX];
## } VCH;

## varchar VCH vch_1; /* typedef referenced */
## varchar struct vch_ vch_2;
##  /* structure tag referenced */
```

- The **varchar** storage class can be used for any type of variable declaration, including external and static variables, and to qualify nested structure members. For example, the following declarations are all legal:

```
## static varchar struct _txt {
##  /* with storage class            "static"*/
##           short      tx_len;
##           char       tx_data[TX_MAX];
## } txt_var,  *txt_ptr,  txt_arr[10];

## struct {
##         char        ename[20];
##         int        eage;
##         varchar struct_txt  ecomments;
##  /* nested in structure */
## } emp;

## typedef short         BUF_SIZE;
## typedef char          BUF[512];

## varchar struct {/* members are typedef'd */
##             BUF_SIZE  len;
##             BUF    data;
## } vchar;
```

Indicator Variables

An indicator variable is a 2-byte integer variable. You can use these in three ways in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a null.
- In a statement that sets data to Ingres, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable to ensure that the associated host variable is large enough to hold the full length of the returned character string.

The base type for an indicator variable must be the integer type **short**. Any type specification built up from **short** is legal, for example:

```
## short    ind;           /* Indicator variable */
## typedef short   IND;

## IND    ind_arr[10];     /* Array of indicators */
## IND    *ind_ptr;        /* Pointer to indicator */
```

Assembling and Declaring External Compiled Forms - VMS only

You can pre-compile your forms in the Visual Forms Editor (VIFRED). By doing so, you save the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file on which to write the MACRO description. After the file is created, you can assemble it into a linkable object module with the VMS command:

macro *filename*

The result of this command is an object file containing a global symbol with the same name as your form.

Before the EQUEL/FORMS **addform** statement can refer to this global object, you must declare it to EQUEL with the following syntax:

globalref int ***_formname_;**

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the variable holding the address of the global object. The *formname* is also used as the title of the form in other EQUEL/FORMS statements. In all statements that use the *formname* as an argument, except for **addform**, you must dereference the name with #.

- The **globalref** storage class associates the object with the external form definition.
- Although you declare *formname* as a pointer, you should *not* precede it with an asterisk when using it in the **addform** statement. The following example shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name. For example:

```
## globalref     int *empform;
## addform empform;       /* The global object */
## display #empform;
    /* The name of the form must be dereferenced
    ** because it is also the name of a variable */
```

Compiling and Declaring External Compiled Forms -UNIX only

You can precompile your forms in VIFRED. This saves the time that would otherwise be required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in C. VIFRED prompts you for the name of the file with the description. After the file is created, you can use the following **cc** command to compile it into linkable object code:

cc -c *filename*

The C compiler usually returns warning messages during this operation. You can suppress these, if you wish, with the **-w** flag on the **cc** command line. This command produces an object file containing a global symbol with the same name as your form.

Before the EQUEL/FORMS statement **addform** can refer to this global object, you must declare it to EQUEL with the following syntax:

extern int ***_formname_;**

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the variable holding the address of the external object. The *formname* is also used as the title of the form in other EQUEL/FORMS statements. In all statements that use the *formname* as an argument, except for **addform**, you must dereference the name with #.
- The **extern** storage class associates the object with the external form definition.
- Although you declare *formname* as a pointer, you should *not* precede it with an asterisk when using it in the **addform** statement.

The following example shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
## extern int *empform;
## addform empform;     /* The global object */
## display #empform;
##   /* The name of the form must be dereferenced */
##   /* because it is also the name of a variable */
```

Concluding Example

The following example demonstrates some simple EQUEL/C declarations:

```
## define MAX_PERSONS 1000
## typedef struct datatypes_ /* Structureof all types */
## {
##    char        d_byte;
##    short       d_word;
##    long        d_long;
##    float       d_single;
##     double         d_double;
##     char         *d_string;
##  } datatypes;

##  datatypes d_rec;

##  char    *dbname = "personnel";
##  char    *formname, *tablename, *columnname;

##  varchar struct {
##        short        len;
##        char        binary_data[512];
##  } binary_chars;
##  enum color {RED, WHITE, BLUE};

##  unsigned int               empid;
##  short int                vac_balance;

##  struct person_ /* Structure with a union */
##  {
##        char     age;
##        long   flags;
##        union
##        {
##         char full_name[30];
##         struct {
##           char       firstname[12],
##                          lastname[18];
##           } name_parts;
##        } person_name;
##  } person, *newperson, person_store[MAX_PERSONS];
```

UNIX

```
##  extern int *empform,*deptform; /* Compiled forms */
```

VMS

```
## globalref int *empform, *deptform; /*Compiled forms*/
```

The Scope of Variables

While the EQUEL precompiler understands the scope of a variable, in programs where this is important, you must ensure that the preprocessor's scoping of the variable coincides with that of the C compiler.

In programs without conflict between multiple variables of the same name declared with different scope, this issue can be ignored. The precompiler does not need to be made aware of scoping information, and it will consider all variables visible to it to belong to one global scope covering the entire source file. Under these circumstances, a second declaration of a particular variable name will generate an error message from the precompiler, and the second declaration will be ignored.

In programs where variable names conflict, or for any other reason scoping becomes an issue, you must observe the following rules to maintain a consistent understanding of scope between the EQUEL precompiler and the C compiler:

- To declare a scope for a particular procedure, or randomly in your source code, use the ## signal with the opening and closing braces. The preprocessor considers all variables declared in these braces as local to that EQUEL scope. For example:

```
       if (error)
##     {
##         int i;  /* i is local */
           EQUEL statement using 'i'
##   }
```

  This is true not only for C blocks, but also for EQUEL statements that are block structured, such as **retrieve**. The braces that delimit EQUEL blocks can also be used as local C blocks and can include variable declarations.

- The above rule holds for fully *enclosed* declarations, such as in the example above or for variables local to a procedure. You can also declare arguments to procedures, but EQUEL may consider these global, depending on where you put the ## signal. For example:

```
        proc1( a )
##      int a;
##      {
            EQUEL statements using 'a'
##      }
```

  In this context, variable "a" is global to the file, which, although legal, may conflict with a later procedure declaration:

```
    proc2( a )
##  char  *a;/* EQUEL complains about redeclaration*/
##  {
          EQUEL statements using 'a'
##  }
```

To solve this problem, put a ## signal on the procedure header and the parameter list. However, it is not necessary to make all of the parameters known to EQUEL, nor is it necessary to make the function return type known. The above problem of proc1 and proc2 having conflicting declarations of "a" could be solved as in the following example:

```
##   proc1( a )
##   int a;
##   {
         EQUEL statements using 'a'
##   }

##   proc2( a )
##   char  *a;      /* EQUEL does not give error */
##   {
         EQUEL statements using 'a'
##   }
```

Note that this does not imply that EQUEL supports function declarations. EQUEL only makes use of the scope information.

- The rules for the scope of a **##define** value are the same as for a variable. If the **##define** statement is in the outermost scope of the file, it is processed like a C **#define** and remains in effect for the whole file. If the **##define** is in a particular EQUEL scope (that is, in a procedure with a ## on the opening and closing braces), then that EQUEL scope is the scope of the defined name.

The following program fragments demonstrate a complete EQUEL/C program syntax:

```
##   /* Global declarations */
##   int      globvar;

     main()
##   {
##       int arg;

          MAIN program uses 'arg' and 'globvar'
##  }

##   proc( arg )
##   int   arg;
##  {
##      float sal;

         C and EQUEL code using 'arg', 'sal'
         and 'globvar'
##  }
```

Variable Usage

C variables declared to EQUEL can substitute for most elements of EQUEL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. The generic uses of host language variables in EQUEL statements are discussed in the *QUEL Reference Guide*. The following discussion covers only the usage issues particular to C language variable types.

You must verify that the statement using the variable is in the scope of the variable's declaration. As an example, the following **retrieve** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idno" as an expression in the where clause:

```
## retrieve (namevar = employee.empname,
##           numvar = employee.empnum) where
##           employee.empnum = idno
```

Simple Variables

The following syntax refers to a simple scalar-valued variable (integer, floating-point, or character string):

simplename

Syntax Notes:

- If you use the variable to send values to Ingres, it can be any scalar-valued variable or **##define** constant, enumerated variable, or enumerated literal.
- If you use the variable to receive values from Ingres, it can only be a scalar-valued variable or enumerated variable.
- Character strings that are declared as:

  char *character_string_pointer**;**

  or:

  char *character_string_buffer***[];**

  are considered scalar-valued variables and should not include any indirection when referenced.
- External compiled forms that are declared as:

UNIX

extern int ***compiled_formname;*

VMS

globalref int **compiled_formname**;*

should not include any indirection when referenced in the **addform** statement:

addform compiled_formname**;**

The following example shows a message handling routine. It passes two scalar-valued variables as parameters: "buffer," a character string, and "seconds," an integer variable.

```
## Print_Message(buffer, seconds)
  ##        char *buffer
  ##        short seconds
  ## {
  ##        message buffer
  ##        sleep seconds
  ## }
```

Array Variables

The following syntax refers to an array variable:

arrayname **[***subscript***]** {**[***subscript***]**}

Syntax Notes:

- You must subscript the variable, because only scalar-valued elements (integers, floating-point, and character strings) are legal EQUEL values.
- When the array is referenced, the EQUEL preprocessor notes the number of indices but does not evaluate the subscript values. Consequently, even though the preprocessor confirms that the correct number of array indirections is used, it accepts illegal subscript values. You must verify that the subscript is legal. For example, the preprocessor accepts both of the following references, even though only the first is correct:

  ```
  ## float salary_array[5];  /* declaration */
     salary_array[0]         /* references */
     salary_array[+-1-+]
  ```

- A character string, declared as an array of characters, is not considered an array and cannot be subscripted in order to reference a single character. In fact, single characters are illegal string values, since all character string values *must* be null-terminated. For example, if the following variable were declared:

  ```
  ## static char abc[3] = {'a', 'b', 'c'};
  ```

  you cannot access the character "a" in an EQUEL statement with the reference:

  ```
  abc[0]
  ```

  To perform such a task, declare the variable as an array of three single character strings:

  ```
  ## static char *abc[3] = {"a","b","c"};
  ```

- Any variable that can be denoted with array subscripting can also be denoted with pointers. This is because the preprocessor only records the number of indirection levels used when referencing a variable. The indirection level is the sum of the number of pointer operators preceding the variable reference name and the number of array subscripts following the name. For example, if a variable is declared as an array:

  ```
  ## int age_set[2];
  ```

  it can be referenced as either an array:

  ```
  age_set[0]
  ```

  or a pointer:

  ```
  *age_set
  ```

  If you use the pointer variant, you must verify that the pointer does not immediately follow a left parenthesis without a separating space, as "(*" is a reserved operator. For example:

  ```
  ## retrieve ( *age_set = e.age )
  ```

  Note the space between the "(" and the "*".

- In an EQUEL statement, do not precede references to elements of an array with the ampersand operator (&) to denote the address of the element.

- Do not subscript arrays of variable addresses that are used with **param** target lists. For example:

  ```
  ##    char      target_list[200];
  ##    char      *addresses[10];

  ##    retrieve   (param(target_list, addresses))
  ```

For more information about parameterized target lists, see Dynamically Built Param Statements in this chapter.

Pointer Variables

The following syntax refers to a pointer variable:

{}*pointername*

Syntax Notes:

- Refer to the variable indirectly, because only scalar-valued elements (integers, floating-point and character strings) are legal QUEL values.

- When the variable is declared, the preprocessor notes the number of preceding asterisks. Later references to the variable must have the same indirection level. The indirection level is the sum of the number of pointer operators (asterisks) preceding the variable declaration name and the number of array subscripts following the name.

- A character string, declared as a pointer to a character, is not considered a pointer and cannot be subscripted in order to reference a single character. As with arrays, single characters are illegal string values, because any character string value *must* be null-terminated. For example, assuming the following declaration:

  ```
  ## char *abc = "abc";
  ```

  you could not access the character "a" with the reference:

  ```
  *abc
  ```

- When you declare external compiled forms in UNIX as:

  extern int ****compiled_formname*;**

  or for VMS as:

  globalref int**compiled_formname*;

  do not include any indirection when referenced in the **addform** statement.

- As with standard C, any variable that you denote with pointer indirection can also be denoted with array subscripting. This is true because the preprocessor only records the number of indirection levels used when referencing a variable. For example, if a variable is declared as a pointer:

  ```
  int *age_pointer;
  ```

  it can be referenced as either a pointer:

  ```
  *age_pointer;
  ```

  or an array:

  ```
  age_pointer[0];
  ```

  If you use the pointer variant, you must verify that the pointer does not immediately follow a left parenthesis without a separating space, as "(*" is a reserved operator. For example:

  ```
  ##    retrieve   (*age_pointer = e.age )
  ```

  Note the space between the "(" and the "*."

  The following example uses a pointer to insert integer values into a database table:

  ```
  ##      int *numptr;
  ##      static int numarr[6] = {1, 2, 3, 4, 5, 0};

          for (numptr = numarr; *numptr; numptr++)
  ##            append items (number = *numptr)
  ```

For information on pointers to structures and members of structures, see Structure Variables in this chapter.

Structure Variables

You cannot use a structure variable as a single entity. Only elementary structure members can communicate with Ingres data. This member must be a scalar value (integer, floating-point, or character string).

Using a Structure Member

The syntax EQUEL uses to refer to a structure member is the same as in C:

*structure***.***member*{**.***member*}

Syntax Notes:

- The structure member the above reference denotes must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and structures, but the last object referenced must be a scalar value. Thus, the following references are all legal in an EQUEL statement, assuming they all translate to scalar values:

```
employee.sal   /* Structure member */
person[3].name /* Member of element of an array */
structure.mem2.mem3.age /* Deeply nested member */
```

- The preprocessor does not check any array elements that are referred to in the structure reference and not at the very end of the reference. Consequently, both of the following references are accepted, even though one must be wrong, depending on whether "person" is an array:

```
person[1].age
person.age
```

- Structure references can also include pointers to structures, denoted by the arrow operator (->). The preprocessor treats the arrow operator exactly like the dot operator and does not check to see that the arrow is used when referring to a structure pointer and that the dot is used when referring to a structure variable. For example, the preprocessor accepts both of the following references to a structure, although only the second one is legal C:

```
## struct
## {
## char *name;
## int  number;
## } people[10], *one_person;

people[i]->name /* Should use the dot operator */
one_person->name
/* Correct use of pointer qualifier   */
```

- In general, the preprocessor supports unambiguous and direct references to structure members, as in the following example:

```
ptr1->struct2.mem3[ind4]->arr5[ind6][ind7]
```

  In this case, the last object denoted, "arr5[ind6][ind7]," must specify a scalar-valued object.

- References to structure variables cannot contain grouping parentheses. For example, assuming "struct1" was declared correctly, the following reference causes a syntax error on the left parenthesis:

  ```
  (struct1.mem2)->num3
  ```

  The only exception to this rule occurs when grouping a reference to the first and main member of a structure by starting the reference with a left parenthesis followed by an asterisk. Note that the two operators, "(" and "*" must be bound together without separating spaces, as in the following example:

  ```
  (*ptr1)->mem2
  ```

  Because "(" and "*" are reserved when not separated by spaces, you must make sure to use the pointer operator (*) correctly after parentheses when dereferencing simple pointers. For more information, see Pointer Variables in this chapter.

- Structures declared with the **varchar** storage class do not reference the structure members. For more information, see Using a Varying Length String Variable (Varchar) in this chapter.

Using an Enumerated Variable (Enum)

The syntax for referring to an enumerated variable or enumerated literal is the same as referring to a simple variable:

```
enum_name
```

Enumerated variables are treated as integer variables when referenced and can be used to retrieve data from and assign data to Ingres. The enumerated literals are treated as integer constants and follow the same rules as integer constants declared with the **##define** statement. Enumerated variables can only be used to assign data to Ingres.

The following program fragment demonstrates a simple example of the enumerated type "color":

```
##  typedef enum {red, white, blue} color;

##  color col_var, *col_ptr;

##  static color col_arr[3] = {blue, white, red};

##  int i;

     /* Mapping from color to string */
     static char *col_to_str_arr[3] =
       {"red","white","blue"};
#    define ctos(c) col_to_str_arr[(int)c];

     /* Fill rows with color array */
     for (i = 0; i < 3; i++)
##     append clr (num = i+1, color = col_arr[i])

     /*
```

```
    ** Retrieve the rows -demonstrating a color variable
    ** and pointer, and arithmetic on a stored color
    ** value. Results are:
    **     [1] blue, red
    **     [2] white, blue
    **     [3] red, white
    */
           col_ptr = &col_arr[0];
##         retrieve (i = clr.num, col_var = clr.color,
##         *col_ptr = clr.color+1)
##         {
              printf("[%d] %s, %s\n", i, ctos(col_var),
              ctos(*col_ptr%3));
##         }
```

Using a Varying Length String Variable (Varchar)

The syntax for referring to a **varchar** variable is the same as referring to a simple variable:

varchar_name

Syntax Notes:

- When using a variable declared with the **varchar** storage class, you cannot reference the two members of the structure individually but only the structure as a whole. This rule differs from the rule that applies to regular structure member referencing. For example, the following declaration and **retrieve** statement are legal:

  ```
  ## varchar struct {
  ##       short        buf_size;
  ##       char        buf[100];
  ## } vch;

  ## retrieve (vch = objects.data)
  ```

  But the following statement will generate an error on the use of the member "buf_size":

  ```
  ## retrieve (vch = objects.data
  ##       vch.buf_size = length(objects.data))
  ```

- When you use the variable to retrieve Ingres data, the 2-byte length field is assigned the length of the data and the data is copied into the fixed length character array. The data is *not* null-terminated. You can use a **varchar** variable to retrieve data in the **retrieve**, **retrieve cursor**, **inquire_ingres**, **getform**, **finalize**, **unloadtable**, **getrow**, and **inquire_frs** statements.

- When you use the variable to set Ingres data, the program must assign the length of the data (in the character array) to the 2-byte length field. You can use a **varchar** variable to set data in the **append**, **replace**, **replace cursor**, **putform**, **initialize**, **loadtable**, **putrow**, **insertrow**, and **set_frs** statements.

Using Indicator Variables

The syntax for referring to an **indicator** variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

host_variable:indicator_variable

Syntax Note:

The indicator variable can be a simple variable, an array element or a structure member that yields a short integer. For example:

```
## short   ind_var, *ind_ptr, ind_arr[5];
      var_1:ind_var
      var_2:*ind_ptr
      var_3:ind_arr[2]
```

Data Type Conversion

A C variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character variables.

Data type conversion occurs automatically for different numeric types, such as from floating-point Ingres database column values into integer C variables, and for character strings, such as from varying-length Ingres character fields into fixed-length C character string variables.

Ingres does *not* automatically convert between numeric and character types. You must use one of the Ingres type conversion functions or a C conversion routine for this purpose.

The following table shows the specific type correspondences for each Ingres data type.

Ingres and C Data Type Compatibility

Ingres Type	C Type
cN	**char [***N*+1**]**
text*(N)*	**char [***N*+1**]**
char*(N)*	**char [***N*+1**]**
varchar(*N***)**	**char [***N*+1**]**
char*(N)(containing ASCII null bytes)*	varchar

Ingres Type	C Type
varchar*(N)(containing ASCII null bytes)*	varchar
i1	short
i2	short
i4	int
f4	float
f8	double
date	char [26]
money	double

The table above shows a choice of two possible correspondences for the **char** and **varchar** Ingres types. If there is any possibility that database columns of these types will hold ASCII null bytes, you should use the **varchar** type in C to represent this data.

Runtime Numeric Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric C variables. It follows the standard type conversion rules. For example, if you assign a **float** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion.

Unsigned integers can be assigned to and retrieved from the database wherever plain integers are used. However, take care when using an unsigned integer whose positive value is large enough to cause the high order bit to be set. Integers such as these are treated as negative numbers in Ingres arithmetic expressions and display as negative numbers by the Forms Runtime system.

The Ingres **money** type is represented as an 8-byte floating-point value, compatible with a C **double**.

Runtime Character Conversion

Automatic conversion occurs between Ingres character string values and C character string variables. The string-valued objects that can interact with character string variables, are:

- Names, such as form and column names
- Database columns of type **c**

- Database columns of type **text**
- Form fields of type **c**

In this context, *character string variables* are not single byte integers declared with the **char** type. They are character string pointers:

char ***character_string_pointer*;**

or references to the character string buffer:

char *character_string_buffer*[*length*];**

Character string pointers are always assumed to be pointing at legal string values or variables. As in any C program, any pointer that has not been initialized to point at a string value will cause either a runtime error, resulting in program failure, or an insidious problem resulting from the overwriting of space in memory.

Also, database columns of type **char** and **varchar** may optionally be handled as strings in EQUEL/C. Several considerations apply when dealing with character string conversions, both to and from Ingres. If your **char** or **varchar** database columns contain ASCII null bytes as data, you should use the C **varchar** storage class rather than C character strings to represent this data.

The following notes apply to data represented in C string variables or constants. For analogous information regarding the **varchar** storage class, see The Varying Length String Type in this chapter.

The conversion of C character string variables used to represent Ingres object names is simple: trailing blanks are truncated from the variables because the blanks make no sense in that context. For example, the string literals "empform " and "empform" refer to the same form and "employees " and "employees" refer to the same database table.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **c** or **char**, a database column of type **text** or **varchar**, or a character-type form field. Ingres pads columns of type **c** and **char** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **text** or **varchar** or in form fields.

Second, the C convention is to *null terminate* character strings, and the Ingres runtime system assumes that all strings *are* null-terminated. For example, the character string "abc" is stored in a C variable as the string literal "abc" followed by the C null character, "\0" requiring four bytes.

Character string variables cannot contain embedded nulls because the runtime system cannot differentiate between embedded nulls and the trailing null terminator. For a description of variables that contain embedded nulls and the C **varchar** storage class, see The Varying Length String Type in this chapter.

When retrieving character data from a Ingres database column or form field into a C character string variable, be sure to always supply enough room in the variable to accommodate the maximum size of the particular object, plus one byte for the C null string terminator. (Consider the maximum size to be the length of the database column or the form field.) If the character string variable is too small to contain the complete string value together with the null character, the runtime system may overwrite other space in memory.

However, if the length of a character string variable is known to the preprocessor, as in the declaration:

```
char        character_string_buffer[fixed_length];
```

then the runtime system copies at most the specified number of characters including the trailing null character. In cases where the fixed length of the variable (less one for the null) is smaller than the data to be copied, the data is truncated. The specified length must be *at least* 2, because one character and the terminating null are retrieved. If the length is exactly 1, the data is overwritten.

Furthermore, take note of the following conventions:

- Data stored in a database column of type **c** or **char** is padded with blanks to the length of the column. The variable receiving such data will contain those blanks, followed by the null character. If the variable is declared with a fixed length known to the preprocessor, the variable receives that many characters, including the terminating null.
- Data stored in a database column of type **text** or **varchar** is not padded with blanks. The character string variable receives only the actual characters in the column, plus the terminating null character. Remember that if **char** or **varchar** database columns contain null characters as data, you should represent them with C variables of the **varchar** storage class, thus avoiding this normal string-handling behavior.
- Data stored in a **character** form field contains no trailing blanks. The character string variable receives only the actual characters in the field, plus the terminating null character.

When inserting character data into an Ingres database column or form field from a C variable, note the following conventions:

- When data is moved from a C variable into a database column of type **c** or **char** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.

- When data is moved from a C variable into a database column of type **text** or **varchar** and the column is longer than the variable, no padding of the column takes place. However, all characters in the variable, including trailing blanks, are inserted. Therefore, you may want to truncate any trailing blanks in character string variables before storing them in columns of these types. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a C variable into a character form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

When comparing character data in an Ingres database column with character data in a C variable, note the following convention:

- When comparing data in **c**, **character**, or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Trailing blanks are significant in **text**. Initial and embedded blanks are significant in **character**, **text**, and **varachar**; they are ignored in **c**.

Caution: The conversion of character string data between Ingres objects and C variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. Care should be taken if using the standard **strcmp** function to test for a change in character data, because blanks are significant in that function.

The Ingres **date** data type is represented as 25-byte character string. Your program should allow 26 characters to accommodate the C null byte at the end.

Using Varchar to Receive and Set Character Data

You can use the C **varchar** storage class to retrieve and set character data. Normally **varchar** variables are used when simple C **char** variables are not sufficient, as when null bytes are embedded in the character data. In those cases the runtime system cannot differentiate between embedded nulls and the null terminator of the string. When using **varchar** variables, the 2-byte length specifier indicates how many bytes are used in the fixed length character array. The runtime system sets this length after data retrieval or by the program before assigning data to Ingres. This length does *not* include a null terminator, as the null terminator is not copied or included in the data. The runtime system copies, at most, the size of the fixed length data buffer into the variable.

You can also use **varchar** variables retrieve character data that does not contain embedded nulls. Here too, no null terminator is included in the data.

Because **varchar** variables never include a null terminator, the program should avoid sending the data member of **varchar** variables to C functions that assume null-terminated strings (such as **strlen** and **strcmp**).

The following program fragment demonstrates the use of the **varchar** storage class for C variables:

```
## static varchar struct vch_ {
##   short      vch_length;
##   char      vch_data[10]; /* Statically initialized */
## } vch_store[3] = {         /* data with nulls */
##      {3, {'1', '2', '3'}},
##      {6, {'1', '2', '3', '\0', '5', '6'}},
##      {8, {'\0', '2', '3', '4', '\0', '6', '7', '8'}}
##      };

## varchar struct vch_ vch_res;

## int i, j;

/*
** Add all three rows of data from table above
** (including nulls). Note that the members of
** the varchar structure are not mentioned.
*/
      for (i = 0; i > 3; i++)
      {
##       append vch (row = i+1; data = vch_store[i])
      }

/*
** Now RETRIEVE the data back. Note that the runtime
** system implicitly assigns to the length field the
** size of the data.
*/
##    retrieve (i = vch.row, vch.res = vch.data)
##    {

   /*
   ** Print the values of each row. Before printing
   ** the values, convert all embedded nulls to the
   ** '?'character for printing. The results are:
    **        [1] '123'
    **    [2] '123?56'
    **    [3] '?234?678'
    */
    for (j = 0; j > vch_res.vch_length; j++)
    {
       if (vch_res.vch_data[j] == '\0')
           vch_res.vch_data[j] = '?';
    }

    /* Note the use of '%/.*s' format here.
    ** This is because varchar data doesn't
    ** contain a null terminator so length is used.
    */

    printf("[%d] '%.*s'\n", i,
          vch_res.vch_length, vch_res.vch_data);

 ## }
```

Dynamically Built Param Statements

EQUEL/C supports a special kind of dynamically built statement called a **param** statement. While the ability to supply names, expression values, and even entire qualifications in the form of host variables, as described in the *QUEL Reference Guide*, provides much dynamic flexibility, **param** statements considerably enhance this flexibility. **Param** statements determine at runtime, not only the names, but also the number and data types of target-list elements. This feature, for example, allows construction of a completely general program that can operate on any table or form that you specify at runtime.

A general restriction on **param** statements is that you cannot use **param** target lists in **repeat** queries.

In EQUEL/C, **param** versions are available for all statements in which:

- Assignments are made between host variables and database columns
- Assignments are made between host variables and form fields (or tablefield columns)

Not only **retrieve**, **append**, and **replace**, but also many forms-related statements such as **getform**, **putform**, **initialize**, **loadtable**, **insertrow**, and several others, have **param** versions.

Consider, again, the reason that these special versions of statements are needed. Non-**param** EQUEL statements, though relatively flexible in terms of substituting variables for expression constants, database and form object names, and entire **where** clauses, are nevertheless fixed at compile time in the number and data type of the objects to or from which assignment is made at runtime. Look at the following non-**param retrieve** statement, for example:

```
##  char   charvar1[100];
##  int    intvar1;
##  float  floatvar1;

##  char   table1[25];
##  char   col1[25], col2[25], col3[25];

/*
** Assignments are made at runtime to all variables
** declared in the two lines immediately above,
** representing names of database objects. Then the
** following RETRIEVE statement gets data from the
** specified table and columns.
*/

##  retrieve (charvar1 = table1.col1,
##    intvar1 = table1.col2,
##    floatvar1 = table1.col3)
```

In this example, host variables represent all components of the target list—the table name and the names of all three columns. What cannot vary in this way of coding, however, is the fact that the **retrieve** statement gets values from exactly three columns, and that you must hard-code the data types of those three columns into the program. **Param** statements allow you to transcend those restrictions.

Syntax of Param Statements

These statements are called **param** statements because of the **param** function in place of its target list. The **param** function has the following syntax:

param *(target_string, var_address_array)*

Thus, for example, a **param retrieve** statement might look like this:

```
##  retrieve (param (targetstr, varaddr))
##  where qual_string
```

The *target_string* is a formatted target list string that can be either a C string variable or a C string constant. Normally it is a variable, since the purpose of this feature is to allow statements to be built at runtime. The *var_address_array* is an array of pointers to which values are assigned at runtime. The elements in this array then hold the addresses of variables of appropriate types to receive or supply data for the table columns or form fields with which the **param** statement interacts.

The *target_string* looks like a regular target list expression, except where a C variable intended to receive or supply data in an assignment would normally appear. In place of these names, the *target_string* contains symbolic type indicators representing the variables. For each of these type indicators appearing in the target list, there must be an address recorded in the corresponding element of the *var_address_array*, beginning with *var_address_array[0]*.

At runtime, EQUEL processes the statement by associating the variable addresses with the type indicators embedded in the *target_string*. Addresses must previously have been placed in the cells of the array in a sequence corresponding to the sequence of type indicators in the *target_string*, such that the statement will find a list of the correct number of C variables of the correct type.

The variable-type indicators can be any of the following:

i2 two-byte integer (**short**)

i4 four-byte integer (**int** or **long**)

f4 four-byte floating-point number (**float**)

f8 eight-byte floating-point number (**double**)

c[N] character string, text

v[N] data stored in a structure of the EQUEL-defined **varchar** storage class/char

In the list above, the length specifier N is optional. For further storage class information, see The Varying Length String Type in this chapter.

In this context, the format indicator must always agree with the C variable that supplies or receives the data. This format does not need to be the same as that of the column where the data is stored in the database. Store data to be retrieved from, or inserted into, table columns of type **date** in character arrays of a length of at least 26 in your program. Items of type **money** should be retrieved into program variables of type float or double.

When you reference ordinary character-string data in a **param** target list, you can use the "c" type indicator with or without specifying the number of characters to be assigned. The optional length specification has the following effect, depending on the kind of statement in which the target list appears:

- In an *input* statement, such as **append** or **putform**, the length specification, N, attached to a "c" type indicator, limits to N the number of bytes actually assigned from the C character string variable to the database or form object. The length specification should not include the null string-termination byte. If N is specified, the string need not be null-terminated.
- In an *output* statement, such as **retrieve** or **getform**, the length specification limits to N the number of bytes of actual data assigned from the database or form object to the C character string variable (this is the number of bytes assigned *before* the null string-terminator is appended). In this context, the length specifier can be useful for preventing the EQUEL runtime system from writing more bytes into a C program variable than the variable has room to hold. In the absence of the length specifier, EQUEL would write into the variable the full length of data located in the column or field and then append the null byte as string terminator.

You must use another type indicator, "v", when referencing data stored in a buffer of the EQUEL-defined **varchar** storage class. (For information about this special storage class, see The Varying Length String Type in this chapter.) The **varchar** class receives and sends data that may contain the ASCII null character as valid data. This applies to both the **char** and **varchar** data types in QUEL. Since the C language ordinarily uses the null character as a string terminator, ordinary string-handling routines are not appropriate for this type of data.

A length specifier, N, can also be used in conjunction with the "v" type indicator. If used, it has the following effect:

- In an *input* statement, such as **append** or **putform**, it is ignored. The count of valid characters, contained in the **varchar** C structure itself, overrides in this case.
- In an *output* statement, such as **retrieve** or **getform**, it limits the number of bytes actually transferred into the data buffer of the varchar C structure.

The following example contains a **param append** statement:

```
main ()
##    {
    /*
    ** Declare variables to be used for supplying data
    ** to the   database
    */

##  char         ch_var[27];
##  int         int_var;
##  double   doub_var;

    /* Declare variables for the PARAM target list, the
    ** array of variable addresses, and the database
    ** table to be used
    */

##  char targlist[100];
##  char *varaddr[10];
##  char tablename[25];

    /* Now assign values to variables in order to set up
    ** the PARAM   statements. In a real application, this
    ** would be done during the process of interacting
    ** with the user, as well as by obtaining
    ** information from system catalogs, or from the
    ** FRS, about the number and data type of table
    ** columns. In this example, the assignments are
    ** hard-coded.
    */

       strcpy (tablename, "employee");

    /* The following target list is for use with
    ** the APPEND statement. Note that the type
    ** indicators appear on the right-hand side of
    ** the assignments. Column names appear on the
    ** left-hand side.
    */

       strcpy (targlist,
            "empname=%c, empnum=%i4, salary=%f8");

    /* The next three statements assign, to an array of
    ** character pointers, the addresses of variables
    ** which will supply data for the APPEND statement.
    ** Because the values being assigned are addresses
    ** of several different types of variables, they
    ** need to be cast to character-pointer type.
    */

       varaddr[0] = (char *) ch_var;
```

```
        varaddr[1] = (char *) &int_var;
        varaddr[2] = (char *) &doub_var;

    /* Next, values are assigned to the data variables
    ** themselves.   Again, in an actual application this
    ** would likely be done by    interacting with the
    ** user.
    */

        strcpy (ch_var, "Swygart, Jane");
        int_var = 332;
        doub_var = 37500.00;

##  ingres "personnel"

##  append to tablename (param (targlist, varaddr))

##  exit

    exit (0);
## }
```

Practical Uses of Param Statements

Most applications do not need **param** statements because programs are usually intended for specific purposes and are based on databases whose designs are known at the time the programs are coded. **Param** statements are crucial mainly for *generic* programs. An example of such a program is QBF, the Ingres user-interface program capable of operating on any database and any table, form, or joindef specified by the user.

It is difficult to illustrate practical examples of **param** statements because in an actual application, you must code to determine the name, number and data type of the objects to be manipulated in a **param** statement target list, in addition to the coding required to obtain or operate on data values. For an extended practical example, see An Interactive Database Browser Using Param Statements in this chapter.

The target string and address array are customarily built from information obtained from various sources: the user, the **formdata** and **tabledata** statements, and the Ingres system catalogs. In an EQUEL/FORMS program, a typical scenario prompts the user for the name of a form to operate on, and then uses the **formdata** and **tabledata** statements to get name and type information about the fields. Subsequently, the various **param** target lists and address arrays the program needs are built using this information. The examples here illustrate only the syntax of the **param** statements themselves, as well as simplified mechanics of setting up their component parts.

The example above, with a **param append**, is typical for an *input* statement, where values are being supplied to the database or form from program variables. Other input statements include **replace**, **initialize**, **putform**, **loadtable**, **putrow**, and so forth.

Output statements are similar, except that the type indicators appear on the left-hand side of the assignment statements in the **param** target list. In these statements, program variables receive data from the database or the form. Output statements include **retrieve**, **getform**, **finalize**, **unloadtable**, **getrow**, and so forth. For the format of the **param** target lists for cursor statements, see Param Versions of Cursor Statements in this chapter.

Indicator Variables in Param Statements

You can code **param** statements to accommodate data assigned to or from nullable columns and form fields. The syntax is analogous to that previously described, with the exception that, in the target string, type indicators are needed in place of both the data variable and the indicator variable. Since indicator variables are always 2-byte integers, you can use the **i2** type indicator used for this purpose. A sample target list of a **param retrieve** statement, including indicator variables, might look like this:

```
targ_list = "%c:%i2=e.empname, %f8:%i2=e.salary";
```

The *var_address_array* corresponding to this target list needs four cells, initialized in the following order:

1. A character-string pointer
2. A pointer to a short
3. A pointer to a double
4. Another pointer to a short

When the **retrieve** statement executes, one or both of the short variables can contain the value -1 if null data were present in that row of the table.

Using the Sort Clause in Param Retrieves

Unlike the non-**param** version of the **retrieve** statement, the **param** version has no application-supplied names for result columns. The non-**param retrieve** uses the same names as for the host variables used to receive the data, but in a **param retrieve** these names are not present in the statement. Only the type indicators are seen by the EQUEL runtime system when the **param retrieve** is executed.

In order to meet the need for result column names in the statement, Ingres generates internal names. If you want to include a **sort** clause in a **param retrieve**, you must use the internally generated result column names as arguments to the **sort** clause. These names are "ret_var1", "ret_var2", and so forth, named sequentially for all the result columns represented by type indicators in the target list. (Ignore null indicators in determining this sequence.) For example, assume a target list as in the previous section:

```
tlist = "%c:%i2=e.empname,%f8:%i2=e.salary";
```

If you want to **retrieve** and **sort by** the result column representing salary, you must supply the internal name "ret_var2" to the **sort** clause:

```
##   retrieve (param(tlist,varaddr))
##   sort by ret_var2:d
```

This sorts by the second result column, in descending order.

Param Versions of Cursor Statements

There are **param** versions for cursor versions of the **retrieve** and **replace** statements. In the case of the cursor retrieve, the **param** target list is used in the **retrieve cursor** statement, not in the **declare cursor** statement. The non-**param retrieve cursor** target list is simply a comma-separated list of C variables corresponding to the result columns identified in the **declare cursor** statement. Therefore, the target string in the **param** version is a comma-separated list of type indicators, optionally with associated type indicators for the null indicator variables.

When you code the **declare cursor** statement for use with the **param** version of **retrieve cursor**, you should take advantage of the fact that the entire target list in **declare cursor** can be replaced by a host string variable. This, in effect, allows the whole retrieve statement in **declare cursor** to be determined at runtime. Then, the components of the param **retrieve cursor** can be built dynamically for the associated **declare cursor** statement.

The target string for a **retrieve cursor** statement might look something like the following:

```
targlist = "%c:%i2,%f8:%i2";
```

This target list is appropriate for a **retrieve cursor** where the associated **declare cursor** retrieved two nullable columns—one character string and one floating-point value.

The **replace cursor** statement also supports a **param** version. Its target list looks the same as in the non-cursor version of **replace**.

The following is a somewhat expanded example, showing both the **declare cursor**, **retrieve cursor**, and **replace cursor**:

```
#     include <stdio.h>
main()
{
      double atof();
      /*
      ** Declare variables to be used for supplying
      ** data to the   database
      */

##    char       ch_var[27];
##    int        int_var;
##    double     doub_var;
##    short      null_ind;
```

```
      /*
      ** Declare variables for the various target
      ** lists and the arrays of variable addresses
      */

##    char   decl_cursor_list[100];
##    char ret_cursor_list[100];
##    char repl_cursor_list[100];
##    char *ret_varaddr[10];
##    char *repl_varaddr[5];
##    int thatsall, ingerror;
      char newsalary[20];

      thatsall = ingerror = 0;

##    ingres "personnel"

      /*
    ** Assign values of target lists for DECLARE CURSOR,
    ** RETRIEVE CURSOR, and REPLACE CURSOR. The second and
    ** third of these have PARAM clauses. The first
    ** doesn't need one, as it transfers no data. In the
    ** target list for RETRIEVE CURSOR, a null indicator
    ** is included for the floating-point value.
    */

       strcpy (decl_cursor_list,
       "employee.empname,employee.age,employee.salary");
       strcpy (ret_cursor_list, "%c26, %i4, %f8:%i2");
       strcpy (repl_cursor_list, "salary=%f8");

    /*
    ** Assign pointer values to the address array
    ** for the RETRIEVE CURSOR statement.
    */

    ret_varaddr[0] = (char *) ch_var;
    ret_varaddr[1] = (char *) &int_var;
    ret_varaddr[2] = (char *) &doub_var;
    ret_varaddr[3] = (char *) &null_ind;

##  declare cursor cursor4 for
##        retrieve (decl_cursor_list)
##  for direct update of (salary)

##  open cursor cursor4

    while (ingerror == 0 && thatsall == 0)
    {

##      retrieve cursor cursor4 (param(ret_cursor_list,
##        ret_varaddr))
##      inquire_ingres (ingerror = errorno,
##      thatsall = endquery)

        /*
        ** If an Ingres error occurred, or if no
        ** more rows found for the cursor, break loop
        */

     if (ingerror  0)
        {
          printf ("Error occurred, exiting ...\n");
          break;
        }
```

```
        if (thatsall == 1)
           {
              printf ("No more rows\n");
               break;
           }

           /* If salary for this record is null, print name
           ** and age, prompt the user to enter the salary,
           ** and replace the value in that row. If salary
           ** is not null, print name, age, and salary.
           */

             if (null_ind == -1)
             {
              printf ("%s, %d\n", ch_var,int_var);
              printf ("Enter Salary: ");
              gets (newsalary);
              doub_var = atof(newsalary);
              if (doub_var  0)
              {
                 repl_varaddr[0] = (char *) &doub_var;

##               replace cursor cursor4
##               (param(repl_cursor_list,repl_varaddr))
              }
           }
           else
           {
               printf ("%s, %d, %10.2f\n", ch_var, int_var,
               doub_var);
           }
         } /* end "while" loop */

## close cursor cursor4

## exit

      exit (0);
}
```

Runtime Error Processing

This section describes a user-defined EQUEL error handler.

Programming for Error Message Output

By default, all Ingres and forms system errors are returned to the EQUEL program, and default error messages are printed on the standard output device. As discussed in the *QUEL Reference Guide*, you can also detect the occurrences of errors in the program by using the **inquire_ingres** and **inquire_frs** statements. (Use **inquire_frs** for checking errors after forms statements. Use **inquire_ingres** for all other EQUEL statements.)

This section discusses an additional technique that enables your program not only to detect the occurrences of errors, but also to suppress the printing of default Ingres error messages, if you choose. The **inquire** statements detect errors but do not suppress the default messages.

This alternate technique entails creating an error-handling function in your program and passing its address to the Ingres runtime routines. This makes Ingres automatically invoke your error handler whenever a Ingres or a forms-system error occurs. You must declare the program error handler as follows:

```
int    funcname (errno)
int    *errno;
{
}
```

You must pass this function to the EQUEL routine **IIseterr()** for runtime bookkeeping using the statement:

```
IIseterr( funcname );
```

This forces all runtime Ingres errors through your function, passing the Ingres error number as an argument. If you choose to handle the error locally and suppress Ingres error message printing, the function should return 0; otherwise the function should return the Ingres error number received.

Avoid issuing any EQUEL statements in a user-written error handler defined to **IIseterr**, except for informative messages, such as **message, prompt, sleep** and **clear screen,** and messages that close down an application, such as **endforms** and **exit**.

The example below demonstrates a typical use of an error function to warn users of access to protected tables. This example passes through all other errors for default treatment.

```
int   locerr( ingerr )
int   *ingerr;
{
# define   TBLPROT   5003
/* error number for protected table */

  if (*ingerr == TBLPROT)
  {
      printf( "You are not authorized for this
      operation.\n" );
      return 0;
  }
  else
  {
      return *ingerr;
  }
}

main()
## {
##  ingres dbname
    IIseterr( locerr );
    ...
##  exit
## }
```

A more practical example would be a handler to catch deadlock errors. For deadlock, a reasonable handling technique in most applications is to suppress the normal error message and simply restart the transaction.

The following EQUEL program executes a Multi-Query Transaction and handles Ingres errors, including restarting the transaction on deadlock.

In this example, a program-defined error handler, rather than the **inquire_ingres** statement, detects Ingres errors. This technique allows the normal Ingres error message to be suppressed in the case of deadlock and the transaction to automatically restart without the user's knowledge.

```
#  define err_deadlock     4700
#  define err_noerror      0

int  ingerr = err_noerror; /* Ingres error */

main()
## {
      int    errproc();
      int    deadlock();

##  ingres "equeldb"      /* set up test data */
##  create item (name=c10, number=i4)
    IIseterr( errproc );

    for(;;)
    {  /* Loop until success or fatal error */
##       begin transaction         /* start MQT*/
##       append to item (name="Neil", number=38)
         if (deadlock())           /* deadlock? */
              continue;            /* yes, try again */
##       replace item (number=39) where item.name="Neil"
         if (deadlock())           /* deadlock? */
              continue;            /* yes, try again */
##       delete item where item.number=38
         if (deadlock())           /* deadlock? */
              continue;            /* yes, try again */
##        end transaction
          break;
    }
##  destroy item
##  exit
##  }

##  /*
##  ** errproc
##  ** - User-defined error routine for Ingres
##  */

int
errproc( errno )
   int  *errno;
{
   ingerr = *errno;        /* set the global flag */

    /*
    ** If we return 0, Ingres will not print a message
    */
    if (*errno == err_deadlock)
        return 0;
    else
        return *errno;
```

```
}

##   /*
##   ** deadlock
##   **  - User-defined deadlock detector
##   **  - If the global error number is not ERR_DEADLOCK,
##   **    it aborts the program and the transaction. If
##   **    the error number is ERR_DEADLOCK, no ABORT
##   **    is necessary because the DBMS will automatically
##   **    ABORT an existing MQT.
##   */

     int
     deadlock()
     ## {
        if (ingerr) {
            if (ingerr == err_deadlock)
        {
            ingerr = err_noerror;
            /* Reset for next time */
          return (1);      /* Program will try again */
         }
         else
         {
           printf ("Aborting -- Error #%d\n",
           ingerr );
##         abort
##         exit
           exit( -1 );
      }
    }
    return 0;
##  }
```

Precompiling, Compiling, and Linking an EQUEL Program

This section describes the EQUEL preprocessor for C and the steps required to precompile, compile, and link an EQUEL program.

Generating an Executable Program

Once you have written your EQUEL program, the preprocessor must convert the EQUEL statements into C code. This section describes the use of the EQUEL preprocessor. Additionally, it describes how to compile and link the resulting code to obtain an executable file.

The EQUEL Preprocessor Command

The following command line invokes the C preprocessor:

eqc {*flags*} {*filename*}

where *flags* are

-d	Adds debugging information to the runtime database error messages EQUEL generates. The source file name, line number, and the erroneous statement itself are printed with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If the **-f** flag is specified without a *filename*, the output is sent to standard output, one screen at a time. If the **-f** flag is omitted, output is given the basename of the input file, suffixed ".c".
-i*N*	Sets integer size to *N* bytes. *N* is 1, 2, or 4. The default is 4.
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename*.**lis**, where *file*name is the name of the input file.
-lo	Like **-l**, but the generated C code also appears in the listing file.
-n.*ext*	Specifies the extension used for filenames in **##include** and **##include inline** statements in the source code. If **-n** is omitted, **include** filenames in the source code must be given the extension ".qc".
-o	Directs the preprocessor not to generate output files for include files. This flag does not affect the translated **include** statements in the main program. The preprocessor generates a default extension for the translated include file statements unless you use the **-o.***ext* flag.
-o. *ext*	Specifies the extension the preprocessor gives to both the translated **include** statements in the main program and the generated output files. If this flag is not provided, the default extension is "c." If you use this flag in combination with the **-o** flag, then the preprocessor generates the specified extension for the translated **include** statements, but does not generate new output files for the **include** statements.
-s	Reads input from standard input and generates C code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Control D** for UNIX or **Control Z** for VMS.

-w Prints warning messages.

-#|-p Generates **#** line directives to the C compiler (by default, they are in comments). This flag can prove helpful when debugging the error messages from the C compiler.

-? Shows the available command line options for **eqc**.

The EQUEL/C preprocessor assumes that input files are named with the extension ".qc". You can override this default by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated C statements with the same name and the extension ".c".

If you enter the command without specifying any flags or a filename, Ingres displays a list of flags available for the command.

The following table presents the options available for **eqc**.

Eqc Command Examples

Command	Comment
eqc file1	Preprocesses "file1.qc" to "file1.c"
eqc -l file2.xc	Preprocesses "file2.xc" to "file2.c" and creates listing "file2.lis"
eqc -s	Accepts input from standard input and writes generated code to standard output
eqc -ffile3.out file3	Preprocesses "file3.qc" to "file3.out"
eqc	Displays a list of flags available for this command

The C Compiler

UNIX

The preprocessor generates C code You can use the UNIX **cc** command to compile this code. All of the **cc** command line options can be used.

The following example preprocesses and compiles the file "test1".

```
eqc test1.qc
cc -c test1.c
```

VMS

The preprocessor generates C code. You should use the VMS **cc** (VAX-11 C) command to compile this code. You can use most of the **cc** command line options. However, you should not use the **g_float** qualifier (to the VAX C compiler) if floating-point values in the file are interacting with Ingres floating-point objects.

The following example preprocesses and compiles the file "test1". Both the EQUEL preprocessor and the C compiler assume the default extensions.

```
eqc test1
cc/list test1
```

Note: Check the Readme file for any operating system specific information on compiling and linking EQUEL/C programs.

Linking an EQUEL Program—UNIX

EQUEL programs require procedures from an Ingres library. The required library is listed in the following examples and must be included in your compile or link command *after* all user modules. The library must be specified in the order shown in the following examples.

Programs without Embedded Forms

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
cc -o dbentry dbentry.o
 $II_SYSTEM/ingres/lib/libingres.a
-lm -lc
```

Note that you must include both the math library and the C runtime library.

Ingres shared libraries are available on some Unix platforms. To link with these shared libraries replace "libingres.a" in your link command with:

```
-L $II_SYSTEM/ingres/lib -linterp.1 -lframe.1 -lq.1 \
     -lcompat.1
```

To verify if your release supports shared libraries check for the existence of any of these four shared libraries in the $II_SYSTEM/ingres/lib directory. For example:

```
ls -l $II_SYSTEM/ingres/lib/libq.1.*
```

Compiling and Linking Precompiled Forms

The technique of declaring a precompiled form to the FRS is discussed in the *Forms-based Application Development Tools User Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in C. VIFRED lets you select the name for the file. After creating the C file this way, you can compile it into linkable object code with the **cc** command:

cc *filename*

The output of this command is a file with the extension ".o". You then link this object file with your program by listing it in the link command, as in the following example, which includes the compiled form "empform.o":

```
cc -o formentry formentry.o
empform.o
 $II_SYSTEM/ingres/lib/libingres.a
-lm -lc
```

Linking an EQUEL Program—VMS

EQUEL programs require procedures from several VMS shared libraries in order to run properly. After preprocessing and compiling an EQUEL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
link dbentry.obj,-
 ii_system:[ingres.files]equel.opt/opt,-
 sys$library:vaxcrtl.olb/library
```

The last line in the link command shown above links in the C runtime library for certain basic C functions, such as **printf**. This line is optional. Use it only if you use those functions in your program.

It is recommended that you do not explicitly link in the libraries referenced in the EQUEL.OPT file. The members of these libraries change with different releases of Ingres. Consequently, you can be required to change your link command files in order to link your EQUEL programs.

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *Forms-based Application Development Tools User Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command:

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry") by listing it in the link command, as in the following example:

```
link formentry,-
  empform.obj,-
  ii_system:[ingres.files]equel.opt/opt,-
  sys$library:vaxcrtl.olb/library
```

Linking an EQUEL Program without Shared Libraries

While the use of shared libraries in linking EQUEL programs is recommended for optimal performance and ease-of-maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by EQUEL are listed in the equel.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
link dbentry,-
  ii_system:[ingres.files]equel.noshare/opt,-
  sys$library:vaxcrtl.olb/library
```

Include File Processing

The EQUEL **include** statement provides a means to include external files in your program's source code. Its syntax is:

include *filename*

Filename is a quoted string constant specifying a file name, a system environment variable in UNIX or a logical name in VMS that points to the file name.

You must use the default extension ".qc" in names of include files unless you override this requirement by specifying a different extension with the **-n** flag of the **eqc** command.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *QUEL Reference Guide*.

The included file is preprocessed and an output file with the same name but with the default output extension ".c" is generated. You can override this default output extension with the **-o***.ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified **include** output file. If you use the **-o** flag with no extension, no output file is generated for the include file. This is useful for program libraries that are using **make** dependencies for UNIX or MMS dependencies for VMS.

If you use both the **-o.***ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the programs. However, it does not generate new output files for the statements.

For example, assume that no overriding output extension is explicitly given on the command line. The EQUEL statement:

```
## include "employee.qc"
```

is preprocessed to the C statement:

```
# include "employee.c"
```

and the file "employee.qc" is translated into the C file "employee.c."

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
## include "MYDECLS";
```

UNIX

The name "MYDECLS" can be defined as a system environment variable pointing to the file "/dev/headers/myvars.qc" by means of the following command at the system level:

```
setenv MYDECLS "/dev/headers/myvars.qc"
```

Assume now that "inputfile" is preprocessed with the command:

```
eqc -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the C statement:

```
# include "/dev/headers/myvars.h"
```

and the C file "/dev/headers/myvars.h" is generated as output for the original include file, "/dev/headers/myvars.qc."

You can also specify include files with a relative path. For example, if you preprocess the file "/dev/mysource/myfile.qc," the EQUEL statement:

```
## include "../headers/myvars.qc"
```

is preprocessed to the C statement:

```
# include "../headers/myvars.c"
```

and the C file "/dev/headers/myvars.c" is generated as output for the original include file, "/dev/headers/myvars.qc."

VMS

The name "mydecls" is defined as a system logical name pointing to the file "dra1:[headers]myvars.qc" by means of the following command at the DCL level:

```
define mydecls dra1:[headers]myvars.qc
```

Assume now that "inputfile" is preprocessed with the command:

```
eqc -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the C statement:

```
# include "dra1:[headers]myvars.h"
```

and the C file "dra1:[headers]myvars.h" is generated as output for the original include file, "dra1:[headers]myvars.qc".

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.qc", the EQUEL statement:

```
## include '[-.headers]myvars.qc'
```

is preprocessed to the C statement:

```
# include "[-.headers]myvars.qc"
```

and the C file "dra1:[headers]myvars.c" is generated as output for the original include file, "dra1:[headers]myvars.qc".

Including Source Code with Labels

Some EQUEL statements generate labels in the output code. If you include a file containing such statements, you must be careful to include the file only once in a given C scope. Otherwise, you may find that the compiler later issues C warning or error messages to the effect that the generated labels are defined more than once in that scope.

The statements that generate labels are the **retrieve** statement and all the EQUEL/FORMS block-type statements, such as **display** and **unloadtable**.

Coding Requirements for Writing EQUEL Programs

The following sections describe coding requirements for writing EQUEL programs.

Comments Embedded in C Output

Each EQUEL statement generates one comment and a few lines of C code. You may find that the preprocessor translates 50 lines of EQUEL into 200 lines of C. This may result in confusion about line numbers when you are debugging the original source code. To facilitate debugging, each group of C statements associated with a particular statement is preceded by a comment corresponding to the original EQUEL source. (Note that only *executable* EQUEL statements are preceded by a comment.) Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file. The **-#** flag to **equel** makes the C comment a C compiler directive, causing any error messages generated by the C compiler to refer to the original file and line number; this may be useful in some cases.

One consequence of the generated comment is that you cannot comment out embedded statements by putting the opening comment delimiter on an earlier line. You have to put the opening comment delimiter on the same line, before the **##** delimiter, to cause the preprocessor to treat the complete statement as a C comment.

Embedding Statements Inside C If Blocks

As mentioned above, the preprocessor may produce several C statements for a single EQUEL statement. However, all the statements generated by the preprocessor are delimited by left and right braces, composing a C block. Thus the statement:

```
if   (!dba)
##   retrieve (passwd = s.#passwd)
##   where s.username = userid
```

produces legal C code, even though the QUEL **retrieve** statement produces more than one C statement. However, two or more EQUEL statements generate multiple C blocks, so you must delimit them yourself, just as you would delimit two C statements in a single **if** block. For example:

```
if   (!dba)
     {
##   message "Confirming your user id"
##   retrieve (passwd = security.#passwd)
##   where security.usrname = userid
     }
```

VMS

Because the preprocessor generates a C block for every EQUEL statement, the VAX C compiler can generate the error "Internal Table Overflow" when a single procedure has a very large number of EQUEL statements and local variables. You can correct this problem by splitting the file or procedure into smaller components.

An EQUEL Statement that Does Not Generate Code

The **declare cursor** statement does not generate any C code. This statement should not be coded as the only statement in C constructs that does not allow *null* statements. For example, coding a **declare cursor** statement as the only statement in a C **if** statement not bounded by left and right braces would cause compiler errors:

```
if    (using_database)
##    declare cursor empcsr for retrieve (employee.ename)
    else
      printf("You have not accessed the database.\n");
```

The code the preprocessor generates is:

```
if (using_database)
else
   printf("You have not accessed the database.\n");
```

which is an illegal use of the C **else** clause.

EQUEL/C Preprocessor Errors

To correct most errors, you may wish to run the EQUEL preprocessor with the listing (**-l**) option on. The listing is sufficient for locating the source and reason for the error.

For preprocessor error messages specific to the C language, see the next section.

Preprocessor Error Messages

The following is a list of error messages specific to the C language:

E_E00001 "The #define statement may be used only with values, not names. Use typedef if you wish to make '%0c' a synonym for a type."

Explanation: The #define directive accepts only integer, floating-point or string literals as the replacement token. You may not use arbitrary text as the replacement token. To define type names you should use typedef. The embedded preprocessor #define is not as versatile as the C #define.

E_E00002 "Cast of #define value is ignored."

Explanation: The preprocessor ignores a cast of the replacement value in a #define statement. Casts, in general, are not supported by the embedded C preprocessor. Remove the cast from the #define statement.

E_E00003 "Incorrect indirection on variable'%0c'. Variable is subscripted, [], or dereferenced, *,%1c time(s) but declared with indirection of%2c."

Explanation: This error occurs when the address or value of a variable is incorrectly expressed because of faulty indirection. For example, the name of an integer array has been given instead of a single array element, or, in the case of character string variables, a single element of the string (that is, a character) has been given instead of a pointer to the string or the name of the array.

Either redeclare the variable with the intended indirection or change its use in the current statement.

E_E00004 "Last component of structure reference'%0c' is illegal."

Explanation: This error occurs when the preprocessor encounters an unrecognized name in a structure reference. The user may have incorrectly typed the name of structure element or may have failed to declare it to the preprocessor.

Check for misspellings in component names and that all of the structure components have been declared to the preprocessor.

E_E00005 Unclosed block – %0x unbalanced left brace(s).

Explanation: The preprocessor reached the end of the file still expecting one or more closing braces (}). Make sure that you have no opening braces in an unclosed character or string constant, or have not accidentally commented out a closing brace. Also remember that the preprocessor ignores #**ifdef** directives, so having several opening braces in alternate paths of an #**ifdef** will confuse the preprocessor.

E_E00006 Unsupported forward declaration of C function "%0c".

Explanation: The preprocessor does not support function declarations. For example, the following declaration will cause this error:

```
##int func():
```

Remove the ## mark from the function declaration.

E_E00007 Unsupported definition of nested C function "%0c". Check for missing closing brace of preceding function.

Explanation: (EQUEL) The preprocessor does not support nested function definitions. This error commonly occurs when the user has omitted the ## mark on the closing brace of the previous function definition.

E_E00008 "Incorrect declaration of C varchar variable is ignored. The members of a varchar structure variable may consist only of a short integer and a fixed length character array."

Explanation: Varchar variables (variables declared with the varchar storage class) must conform to an exact varying length string template so that Ingres can map to and from them at runtime. The length field must be exactly two bytes (derived from a short), and the character string field must be a single-dimensioned C character array. The varchar clause must be associated with a variable declaration and not with a type definition or structure tag declaration.

Check the varchar structure declaration. Make sure that both structure members are declared properly.

E_E00009 "Missing'=' in the initialization part of a C declaration."

Explanation: The preprocessor allows automatic initialization of variables and expects the regular C syntax. Insert an equals sign between the variable and the initializing value.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application using two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Department information is stored in program variables. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo (the "toberesolved" database table, described below) to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second is for the Employee table. The **create** statements used to create the tables are shown below. The cursors retrieve all the information in their respective tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, both from the Department table and the Employee table, is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the EQUEL statements. The program illustrates table creation, multi-query transactions, all cursor statements and direct updates. For purposes of brevity, error handling on data manipulation statements is simply to close down the application.

The following two **create** statements describe the Employee and Department database tables:

```
##  create dept
##    (name  = c12,  /* Department name */
##     totsales  = money,  /* Total sales */
##     employees = i2)  /* Number of employees */

##  create employee
##     (name  = c20,  /* Employee name */
##      age  = i1,  /* Employee age */
##      idno  = i4,  /* Unique employee id */
##      hired  = date,  /* Date of hire */
##      dept  = c10,  /* Employee department */
##      salary  = money)  /* Yearly salary */

/* Global variable set in error handler */
int  is_error = 0;

/*
** Procedure:    MAIN
** Purpose:    Main body of the application. Initialize the
** database process each department, and terminate the session.
** Parameters:
** None
*/
```

```
main()
{
   printf("Entering application to process expenses.\n");
   Init_Db();
   Process_Depts();
   End_Db();
   printf("Successful completion of application.\n");
}

/*
** Procedure:  Init_Db
** Purpose:  Initialize the database.
** Start up the database, and abort if an error.
** Before processing employees, create the table
** for employees who lose their department,
** "toberesolved". Initiate the multi-statement
** transaction.
** Parameters:
** None
*/

## Init_Db()
## {
## char  err_text[257];
   int   Error_Proc();

## ingres personnel;

/* Inform Ingres runtime system about error handler */
  IIseterr(Error_Proc);

printf ("Creating \"To_Be_Resolved\" table.\n");
## create toberesolved
##  (name  = c20,
##  age  = i1,
##  idno  = i4,
##  hired = date,
##  dept  = c10,
##  salary  = money)

if (is_error)
  {
##  inquire_ingres (err_text = errortext)
    printf("Fatal error on creation:\n%s", err_text);
##  exit
    exit(-1);
 }

## begin transaction
## }

/*
** Procedure:  End_Db
** Purpose:  Close off the multi-statement transaction and
** access to the database after successful completion
** of the application.
** Parameters:
** None
*/

## End_Db()
## {
## end transaction
## exit
## }
```

```
/*
** Procedure:  Process_Depts
** Purpose:  Scan through all the departments, processing each one.
** If the department has made less than $50,000 in sales,
** then the department is dissolved. For each department
** process all the employees (they may even be moved to
** another table). If an employee was terminated, then
** update the department's employee counter. No error
** checking is done for cursor updates.
** Parameters:
** None
*/

## Process_Depts()
## {
## struct dpt {   /* Corresponds to the "dept" table */
##  char        name[13];
##  double      totsales;
##  short       employees;
##  } dpt;
##  int   no_rows = 0;   /* Cursor loop control */
##   define min_dept_sales 50000.00   /* Min sales of department */
##  short   emps_term = 0;   /* Employees terminated */
    short   deleted_dept;   /* Was the dept deleted? */
    char    *dept_format;   /* Formatting value */

is_error = 0;   /* Initialize error flag */
## range of d IS dept
## declare cursor deptcsr for
##  retrieve (d.name, d.totsales, d.employees)
##  for direct update of (name, employees)

## open cursor deptcsr
   if (is_error)
    Close_Down();

while (!no_rows)
   {
   is_error = 0;
## retrieve cursor deptcsr
## (dpt.name, dpt.totsales, dpt.employees)
## inquire_equel (no_rows = endquery)

if (!no_rows)
   {
    /* Did the department reach minimum sales? */
     if (dpt.totsales < min_dept_sales)
     {
##    delete cursor deptcsr
      /* If error occurred in deleting row, close down */
       if (is_error)
          Close_Down();
       deleted_dept = 1;
       dept_format = " -- DISSOLVED --";
       }
        else
       {
       deleted_dept = 0;
       dept_format = "";
       }

/* Log what we have just done */
    printf( "Department: %14s, Total Sales: %12.3f %s\n",
     dpt.name, dpt.totsales, dept_format );

/* Now process each employee in the department */
```

```
    Process_Employees( dpt.name, deleted_dept, &emps_term );

/* If some employees were terminated, record this fact */
    if (emps_term > 0 && !deleted_dept)
##    replace cursor deptcsr
##     (employees = dpt.employees - emps_term)

/* If error occurred in update, close down application */
    if (is_error)
      Close_Down();
   }
  }

## close cursor deptcsr
##  }

/*
** Procedure:  Process_Employees
** Purpose:  Scan through all the employees for a particular
** department. Based on given conditions the employee
** may be terminated, or given a salary reduction.
** 1. If an employee was hired since 1985 then the
** employee is terminated.
** 2. If the employee's yearly salary is more than
** the minimum company wage of $14,000 and the
** employee is not close to retirement (over 58
** years of age), then the employee takes a 5%
** salary reduction.
** 3. If the employee's department is dissolved and
** the employee is not terminated, then the employee
** is moved into the "toberesolved" table.
** Parameters:
** dept_name  -  Name of current department.
** deleted_dept  -  Is current department being dissolved?
** emps_term  - Set  locally to record how many
** employees were terminated for the
** current department.
*/

## Process_Employees( dept_name, deleted_dept, emps_term )
##  char    *dept_name;
    short   deleted_dept;
    short   *emps_term;
## {
## struct emp { /* Corresponds to "employee" table */
##  char     name[21];
##  short    age;
##  int      idno;
##  char     hired[26];
##  float    salary;
##  int      hired_since_85;
## } emp;
##  intno_rows = 0;  /* Cursor loop control */
##  define min_emp_salary 14000.00   /* Minimum employee salary */
##  define nearly_retired 58
##  define salary_reduc 0.95
  char *title;       /* Formatting values */
  char *description;

is_error = 0;    /* Initialize error flag */
  /*
  ** Note the use of the Ingres function to find out who was hired
  ** since 1985.
  */
## range of e is employee
## declare cursor empcsr for
```

```
##  retrieve (e.name, e.age, e.idno, e.hired, e.salary, res =
##  int4(interval("days", e.hired-date("01-jan-1985"))))
##  where e.dept = dept_name
##  for direct update of (name, salary)

## open cursor empcsr
   if (is_error)
    Close_Down();

*  emps_term = 0;   /* Record how many */
   while (!no_rows)
   {
   is_error = 0;
## retrieve cursor empcsr (emp.name, emp.age, emp.idno,
## emp.hired, emp.salary, emp.hired_since_85)
## inquire_equel (no_rows = endquery)

if (!no_rows)
  {
     if (emp.hired_since_85 > 0)
     {
##   delete cursor empcsr
     if (is_error)
       Close_Down();
     title = "Terminated:";
     description = "Reason: Hired since 1985.";
     (*emps_term)++;
     }
     else
     {
       /* Reduce salary if not nearly retired */
        if (emp.salary > MIN_EMP_SALARY)
        {
          if (emp.age < nearly_retired)
          {
##        replace cursor empcsr
##        (salary = salary * salary_reduc)
        if (is_error)
          Close_Down();
        title = "Reduction: ";
        description = "Reason: Salary.";
       }
       else
       {
         /* Do not reduce salary */
         title = "No Changes:";
         description = "Reason: Retiring.";
       }

}
    else   /* Leave employee alone */
    {
       title = "No Changes:";
       description = "Reason: Salary.";
    }

/* Was employee's department dissolved ? */
      if (deleted_dept)
      {
##      append to toberesolved (e.all)
##       where e.idno = emp.idno
        if (is_error)
         Close_Down();
##      delete cursor empcsr
      }
    }
```

```
/* Log the employee's information */
     printf(" %s %6d, %20s, %2d, %8.2f; %s\n",
      title, emp.idno, emp.name, emp.age, emp.salary, description);
   }
 }

## close cursor empcsr
   is_error = 0;
## }

/*
** Procedure:  Close_Down
** Purpose:  If an error occurs during the execution of an
** EQUEL statement,    the error handler sets a flag which
** may cause this routine to be called. For simplicity,
** errors cause the current transaction to be aborted and
** the application to be closed down.
*/

## Close_Down()
## {
## char err_text[257];
## inquire_ingres (err_text = ERRORTEXT)
   printf("Closing down because of database
   error:\n%s",         err_text);
## abort
## exit
   exit(-1);
## }

/*
** Procedure:  Error_Proc
** Purpose:  Process Ingres errors
**  Set global "is_error" flag, allowing appropriate action
**  after individual database statements. Return 0 so that
**  Ingres runtime system will suppress error messages.
** Parameters:
**  ingerr - Pointer to integer containing
**  ingres error number.
*/

int
Error_Proc(ingerr)
int    *ingerr;
{
  is_error = 1;
  return 0;
}
```

The Employee Query Interactive Forms Application

This section contains a sample EQUEL/FORMS application that uses a form in **query** mode to view a subset of the Employee table in the Personnel database. An Ingres query qualification is built at runtime using values entered in fields of the form "empform."

The objects used in this application are:

Object	Description
personnel	The program's database environment.
employee	A table in the database, with six columns: name (**c20**) age (**i1**) idno (**i4**) hired (**date**) dept (**c10**) salary (**money**).
empform	A VIFRED form with fields corresponding in name and type to the columns in the Employee database table. The Name and Idno fields are used to build the query and are the only updatable fields. "Empform" is a compiled form.

A **display** statement drives the application. This statement allows the runtime user to enter values in the two fields that build the query. The **Build_Query** and **Exec_Query** procedures make up the core of the query that is run as a result. Note the way the values of the query operators determine the logic that builds the **where** clause in **Build_Query**. The **retrieve** statement encloses a **submenu** block that allows the user to step through the results of the query.

The retrieved values are not updated, but any employee screen can be saved in a log file using the **printscreen** statement in the **save** menu item.

The following **create** statement describes the format of the Employee database table:

```
##   create employee
##     (name   = c20,    /* Employee name */
##      age    = i1,     /* Employee age */
##      idno   = i4,     /* Unique employee id */
##      hired  = date,   /* Date of hire */
##      dept   = c10,    /* Employee department */
##      salary = money)  /* Annual salary */

/*
** Procedure: MAIN
** Purpose:   Entry point into Employee Query application.
```

```
*/

## main()
## {

##
    extern int *empfrm;         /* Compiled form - UNIX */

     /* For VMS the compiled form is declared using the statement
     ** 'globalref int *empform;'
     */
    char where_clause[101];    /* For WHERE clause qualification */

     /*
     **       Initialize global WHERE clause qualification buffer
     **      to be an Ingres default qualification that is
     **      always true
     */

    strcpy (where_clause, "1=1");

##  forms
##  message "Accessing Employee Query Application . . ."
##  ingres personnel

##  range of e is employee

##  addform empfrm

##  display #empfrm query
##  initialize

##  activate menuitem "Reset"
##  {
##    clear field all
##  }

##  activate menuitem "Query"
##
      /* Verify validity of data */
##    validate
      Build_Query(where_clause);
      Exec_Query(where_clause);
##  }

##  activate menuitem "LastQuery"
##  {
      Exec_Query(where_clause)
##  }

##  activate menuitem "End"
##  {
##    breakdisplay
##  }
##  finalize

##  clear screen
##  endforms
##  exit

## } /* main */

/*
** Procedure:   Build_Query
** Purpose:     Build an Ingres query from the values in the
**              'name' and 'idno' fields in 'empfrm.'
```

```
** Parameters:  where_clause
**              Pointer to array for building WHERE clause. */**

Build_Query(where_clause)
char      *where_clause

## {
##   char ename[21];      /* Employee name */
##   int  eidno;          /* Employee id */
##   int  name_op, id_op; /* Query operators */

/*  Query operator table maps integer values to string
**   query operators static char
**   *opertab[] = {'=', '!=', '<', '>', '<=', '>='};

## getform #empfrm
##        (Ename = name, nameop = getoper(name),
##         Eidno = idno, idop = getoper(idno))

   /*  Fill in the WHERE clause */
   if (name_op == 0 && id_op == 0)
   {
          strcpy (where_clause,'1=1');
   }
   else
   {
          if (name_op !=0 && id_op != 0)
          {
            /* Query on both fields */
            sprintf (where_clause, "e.name %s \"%s\"
            and e.idno %s %d",
            opertab[name_op -1], ename,
            opertab[id_op -1], eidno);
          }
          else if (name_op != 0)
          {
            /* Query on the "name" field */
            sprintf (where_clause, "e.name %s \"%s\"",
            opertab[name_op -1], ename);
          }
          else
          {
             /* Query on the '"idno" field */
             sprintf (where_clause, "e.idno %s %d",
             opertab[id_op -1], eidno);
          }
      }

## }

/*
**
**  Procedure:   Exec_Query
**  Purpose:    Given a query buffer defining a WHERE clause, issue
**              a RETRIEVE to allow the runtime user to browse the
**              employee found with the given qualification.
** Parameters:   where_clause
**               - Contains WHERE clause qualification.
**
*/


## Exec_Query(where_clause)
##  char *where_clause;
## {
```

```
   /* Employee data */

## char    ename[21];
## short   eage;
## int     eidno;
## char    ehired[26];
## char    edept[11];
## float   esalary;
## int     rows; /*Were rows found? */

## retrieve (ename = e.name, eage = e.age, eidno = e.idno,
##     ehired = e.hired, edept = e.dept, epay = e.salary)
##     where where_clause
## {
##     /* put values on to form and display them */
##     putform empfrm
##        (name = ename, age = eage, idno = eidno, hired = ehired,
##        dept = edept, salary = epay)
##        redisplay
##        submenu
##        activate menuitem "Next"
##    {
         /*
         **  Do nothing, and continue with the RETRIEVE loop. The
         **  last one will drop out.
         */
##    }

##    activate menuitem "Save"
##    {
##      /*  Save screen data in log file */
##      printscreen (file = 'query.log')
         /*  Drop through to next employee */
##    }

##    activate menuitem "End"
##    {
         /*  Terminate the RETRIEVE loop */
##      endretrieve
##    }
## }

## inquire_equel (rows = ROWCOUNT)
   if (rows == 0)
   {
##     message "No rows found for this query"
   else
   {
##      clear field all
##      message "Reset for next query"
   }

##   sleep 2

## }
```

The Table Editor Table Field Application

This EQUEL/FORMS application uses a table field to edit the Person table in the Personnel database. It allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate their use and their interaction with an Ingres database.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**c20**) age (**i2**) number (**i4**) Note: number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table file in the form, with two columns: name (**c20**) age (**i4**). When initialized, the table field includes the hidden number (**i4**) column.

When the application begins, a **retrieve** statement is issued to load the table field with data from the Person table. Once the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated, or deleted. When finished, the values are unloaded from the table field, and, in a multi-statement transaction, the user's updates are transferred back into the Person table.

The following **create** statement describes the format of the Person database table:

```
## create person
##   (name  = c20,   /* Person name */
##    age  = i2,     /* Age */
##    number = i4)   /* Unique id number */

/*
** Global structure pers corresponds to "person" table
*/
##  struct {
## char pname[21]; /* Full name (with C null) */
## int  page; /* Age of person */
## int  pnumber; /* Unique person number */
```

```
## int  maxid; /* Max person id number */
##  }pers;

/*
** Procedure: MAIN
** Purpose: Entry point into Table Editor program.
*/

## main()
## {
/*  Table field row states */
# define stundef 0 /* Empty or undefined row */
# define stnew  1 /* Appended by user */
# define stunchanged 2 /* Loaded by program - not updated */
# define stchange 3 /* Loaded by program - since changed */
# define stdelete 4 /* Deleted by program */

/*  Table field entry information */
## int state; /* State of data set entry */
## int record;  /* Record number */
## int lastrow; /* Last row in table field */

/*  Utility buffers */
## char msgbuf[256]; /* Message buffer */
## char respbuf[256]; /* Response buffer*/

/*  Status variables */
## int update_error; /* Update error from database */
## int update_rows; /* Number of rows updated */
   int xact_aborted; /* Transaction aborted */

/*  Start up Ingres and the FORMS system */
## ingres "personnel"

## forms

/*  Verify that the user can edit the "person" table */
## prompt noecho ("Password for table editor: ", respbuf)

if (strcmp(respbuf, "MASTER_OF_ALL") != 0)
    {
## message "No permission for task. Exiting . . ."
## endforms
## exit
   exit(-1);
 }

##message "Initializing Person Form . . ."

##range of p is person

##forminit personfrm

/*
 ** Initialize "persontbl" table field with a data set in FILL
 ** mode so that the runtime user can append rows. To keep track
 ** of events occurring to original rows that will be loaded into
 ** the table field, hide the unique person number.
 */
##  inittable personfrm persontbl fill (number = i4)

Load_Table();

## display personfrm update
## initialize
```

```
## activate menuitem "Top"
## {
    /*
    ** Provide menu, as well as the system FRS key to scroll
    ** to both extremes of the table field.
    */
##  scroll personfrm persontbl to 1
## }

## activate menuitem "Bottom"
## {
##  scroll personfrm persontbl to end /* Forward */
## }

## activate menuitem "Remove"
## {
    /*
    ** Remove the person in the row the user's cursor is on.
    */
##  deleterow personfrm persontbl /* Record later */
## }

## activate menuitem "Find"
## {
    /*
    ** Scroll user to the requested table field entry.
    ** Prompt the user for a name, and if one is typed in
    ** loop through the data set searching for it.
    */
##  prompt ("Person's name : ", respbuf)
    if (respbuf[0] == '\0')
##     resume field persontbl

##  unloadtable personfrm persontbl
##   (pers.pname = name, record = _record, state = _state)
##  {
      /* Do not compare with deleted rows */
      if ((strcmp(pers.pname, respbuf) == 0) && (state != stDELETE))
      {
##      scroll personfrm persontbl TO record
##      resume field persontbl
      }

##   }

/* Fell out of loop without finding name */ sprintf(msgbuf,
    "Person \"%s\" not found in table [HIT RETURN] ", respbuf);
##    prompt noecho (msgbuf, respbuf)
## }

## activate menuitem "Exit"
## {
##  validate field persontbl
##  breakdisplay
## }
## finalize

/*
 ** Exit person table editor and unload the table field. If any
 ** updates, deletions or additions were made, duplicate these
 ** changes in the source table. If the user added new people we
 ** must assign a unique person id before returning it to
 ** the table. To do this, increment the previously saved
 ** maximum id number with each insert.
 */
```

```
/*  Do all the updates in a transaction */
## begin transaction

update_error = 0;
 xact_aborted = 0;

## message "Exiting Person Application . . .";
## unloadtable personfrm persontbl
##  (pers.pname = name, pers.page = age,
##  pers.pnumber = number, state = _state)
##{

/* Appended by user. Insert with new unique id */
   if (state == stnew)
   {
   pers.maxid = pers.maxid + 1;
## repeat append to person (name = @pers.pname,
##        age = @pers.page,
##        number = @pers.maxid)
   }
   /* Updated by user. Reflect in table */
   else if (state == stchange)
   {
## repeat replace p (name = @pers.pname, age = @pers.page)
##  where p.number = @pers.pnumber
      }
      /*
      ** Deleted by user, so delete from table. Note that only
      ** original rows are saved by the program, and not rows
      ** appended at runtime.
      */
      else if (state == stdelete)
      {
##      repeat delete from p where p.number = @pers.pnumber
      }
      /* Else UNDEFINED or UNCHANGED - No updates */

    /*
      ** Handle error conditions -
      ** If an error occurred, then abort the transaction.
      ** If no rows were updated then inform user, and
      ** prompt for continuation.
      */
##    inquire_ingres (update_error = errorno, update_rows=rowcount)
      if (update_error) /* Error */
      {
##      inquire_equel (msgbuf = errortext)
##      abort
        xact_aborted = 1;
##      endloop
      }
      else if (!update_rows)
      {
         sprintf(msgbuf,
           "Person \"%s\" not updated. Abort all updates? ",
           pers.pname);
##       prompt (msgbuf, respbuf)
         if (respbuf[0] == 'Y' || respbuf[0] == 'y')
         {
##          abort
            xact_aborted = 1;
##          endloop
         }
      }

##  }
```

```
if (!xact_aborted)
## end transaction /* Commit the updates */

##  endforms /* Terminate the FORMS and Ingres */
##  exit

    if (update_error)
    {
       printf( "Your updates were aborted because of error:\n" );
       printf( msgbuf );
       printf( "\n" );
    }

## } /* Main Program */

/*
** Procedure: Load_Table
** Purpose: Load the table field from the "person" table. The
** columns "name" and "age" will be displayed, and
** "number" will be hidden.
** Parameters:
** None
** Returns:
** Nothing
*/

## Load_Table()
## {
  /* Set up error handling for loading procedure */

message "Loading Person Information . . ."

/*  Fetch the maximum person id number for later use */
##         retrieve (pers.maxid = max(p.number))

/*  Fetch data, and load table field */
## retrieve (pers.pname = p.name, pers.page = p.age,
##  pers.pnumber = p.number)
## {
##  loadtable personfrm persontbl
##   (name = pers.pname, age = pers.page,
##     number = pers.pnumber)
## }

##  } /* Load_Table */
```

The Professor-Student Mixed Form Application

This EQUEL/FORMS application lets the user browse and update information about graduate students who have a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**c25**) pdept (**c10**). See its create statement below for a full description.
student	A database table with seven columns: sname (**c25**) sage (**i1**) sbdate (**c25**) sgpa (**f4**) sidno(**i1**) scomment (**text(200)**) sadvisor (**c25**). See the create statement below for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has the pname and pdept fields, which correspond to the information in the Professor table, and studenttbl table field. The pdept field is display-only. "Masterfrm" is a compiled form.
studenttbl	A table field in "masterfrm" with two columns, sname and sage. When initialized, it also has five more hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the sgpa, scomment, and sadvisor fields are updatable. All other fields are display-only. "Studentfrm" is a compiled form.
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the "studentfrm" form and the "studenttbl" table field.

The program uses the "masterfrm" as the general-level master entry, in which data can only be retrieved and browsed, and the "studentfrm" as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the pname (professor name) field and then selects the **Students** menu operation. The operation fills the displayed and hidden columns of the studenttbl table field with detailed information of the students reporting to the named professor. The user can then browse the table field (in **read** mode), which displays only the names and ages of the students. To request more information about a specific student, select the **Zoom** menu operation. This operation displays the form "studentfrm." The fields of "studentfrm" are filled with values stored in the hidden columns of "studenttbl." The user can make changes to three fields (sgpa, scomment and sadvisor). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

The following two **create** statements describe the Professor and Student database tables:

```
## create student /* Graduate student table */
##  (sname = c25, /* Name */
##   sage = i1, /* Age */
##   sbdate = c25, /* Birth date */
##   sgpa = f4, /* Grade point average */
##   sidno = i4, /* Unique student number */
##   scomment = text(200), /* General comments */
##   sadvisor = c25) /* Advisor's name */
## create professor /* Professor table */
##  (pname = c25, /* Professor's name */
##   pdept = c10) /* Department */

/*
** GLOBAL declaration
** grad student record maps to database table
*/
## struct {
##  char   sname[26];
##  short   sage;
##  char   sbdate[26];
##  float   sgpa;
##  int   sidno;
##  char   scomment[201];
##  char   sadvisor[26];
## } grad;

/*
** Procedure: DECLARE FORMS
*/

## extern int *masterfrm; /* Compiled forms - UNIX */
## extern int *studentfrm;
  /* For VMS, to declare the compiled form use the statements
  ** 'globalref int *masterfrm;' and 'globalref int *studentfrm;'
  */

  /*
  ** Procedure: MAIN
  ** Purpose:  Start up program and call Master driver.
  */
```

```
main()
{
    /* Start up Ingres and the FORMS system */
## forms
## message "Initializing Student Administrator . . ."

## ingres personnel
## range of p is professor, s is student

Master();

## clear screen
## endforms
## exit
}
/*
** Procedure:  Master
** Purpose:  Drive the application, by running "masterfrm", and
** allowing the user to "zoom" into a selected student.
** Parameters:
** None - Uses the global student "grad" record. */

Master()
## {
 /* Professor info maps to database table */
## struct {
## char pname[26];
## char pdept[11];
##} prof;

/* Useful forms system information */
## int lastrow; /* Lastrow in table field */
## int istable; /* Is a table field? */

/* Local utility buffers */
## char msgbuf[100]; /* Message buffer */
## char respbuf[256]; /* Response buffer */
## char old_advisor[26]; /* Old advisor before ZOOM */

/* Externally compiled master form - UNIX */
## extern int *masterfrm;
/* For VMS use 'globalref int *masterfrm;' */

## addform masterfrm

/*
 ** Initialize "studenttbl" with a data set in READ mode.
 ** Declare hidden columns for all the extra fields that
 ** the program will display when more information is
 ** requested about a student. Columns "sname" and "sage"
 ** are displayed, all other columns are hidden, to be
 ** used in the student information form.
 */
## inittable #masterfrm studenttbl read
##  (sbdate = c25,
##  sgpa = float4,
##  sidno = integer4,
##  scomment = c200,
##  sadvisor = c20)

## display #masterfrm update

## initialize
## {
##  message "Enter an Advisor name . . ."
```

```
##  sleep 2
## }

## activate menuitem "Students", FIELD "pname"
## {
   /* Load the students of the specified professor */
## getform (prof.pname = pname)

/* If no professor name is given then resume */
   if (prof.pname[0] == '\0')
##   resume field pname

/*
  ** Verify that the professor exists. Local error
  ** handling just prints the message, and continues.
  ** We assume that each professor has exactly one
  ** department.
  */
  prof.pdept[0] = '\0';
## retrieve (prof.pdept = p.pdept)
##  where p.pname = prof.pname

if (prof.pdept[0] == '\0')
  {
     sprintf(msgbuf,
      "No professor with name \"%s\" [RETURN]", prof.pname);
##   prompt noecho (msgbuf, respbuf)
##   clear field all
##   resume field pname
  }

/* Fill the department field and load students */
##  putform (pdept = prof.pdept)
##  redisplay  /* Refresh for query */

Load_Students(prof.pname);

##  resume field studenttbl

##}     /* "Students" */

## activate menuitem "Zoom"
## {

/*
 ** Confirm that user is on "studenttbl", and that
 ** the table field is not empty. Collect data from
 ** the row and zoom for browsing and updating.
 */
##  inquire_frs field #masterfrm (istable = table)

if (istable == 0)
  {
##   prompt noecho
##     ("Select from the student table [RETURN]", respbuf)
##   resume field studenttbl
  }

##   inquire_frs table #masterfrm (lastrow = lastrow)

if (lastrow == 0)
  {
##   prompt noecho ("There are no students [RETURN]", respbuf)
##   resume field pname
  }
```

```
/* Collect all data on student into global record */
## getrow #masterfrm studenttbl
##   (grad.sname = sname,
##    grad.sage = sage,
##    grad.sbdate = sbdate,
##    grad.sgpa = sgpa,
##    grad.sidno = sidno,
##    grad.scomment = scomment,
##    grad.sadvisor = sadvisor)

/*
  ** Display "studentfrm", and if any changes were made
  ** make the updates to the local table field row.
  ** Only make updates to the columns corresponding to
  ** writable fields in "studentfrm". If the student
  ** changed advisors, then delete this row from the
  ** display.
  */
  strcpy(old_advisor, grad.sadvisor);
    if (Student_Info_Changed())
    {
      if (strcmp(old_advisor, grad.sadvisor) != 0)
##     deleterow #masterfrm studenttbl
      else
##     putrow #masterfrm studenttbl
##      (sgpa = grad.sgpa,
##       scomment = grad.scomment,
##       sadvisor = grad.sadvisor)
  }

## }  /* "Zoom" */

## activate menuitem "Exit"
## {
##  breakdisplay
## }  /* "Exit" */

## finalize

## }  /* Master */

/*
** Procedure:  Load_Students
** Purpose:  Given an advisor name, load into the "studenttbl"
** table field all the students who report to the
** professor with that name.
** Parameters:
** advisor - User specified professor name.
** Uses the global student record.
*/

Load_Students(advisor)
##    char  *advisor;
{
   /*
   ** Clear previous contents of table field. Load the table
   ** field from the database table based on the advisor name.
   ** Columns "sname" and "sage" will be displayed, and all
   ** others will be hidden.
   */
## message "Retrieving Student Information . . ."
## clear field studenttbl

## retrieve
##  (grad.sname = s.sname,
##   grad.sage = s.sage,
```

```
##   grad.sbdate = s.sbdate,
##   grad.sgpa = s.sgpa,
##   grad.sidno = s.sidno,
##   grad.scomment = s.scomment,
##   grad.sadvisor = s.sadvisor)
##  where s.sadvisor = advisor
## {
##  loadtable #masterfrm studenttbl
##  (sname = grad.sname,
##   sage = grad.sage,
##   sbdate = grad.sbdate,
##   sgpa = grad.sgpa,
##   sidno = grad.sidno,
##   scomment = grad.scomment,
##   sadvisor = grad.sadvisor)
##}

} /* Load_Students */

/*
** Procedure:  Student_Info_Changed
** Purpose:  Allow the user to zoom into the details of a
** selected student. Some of the data can be updated
** by the user. If any updates were made, then reflect
** these back into the database table. The procedure
** returns TRUE if any changes were made.
** Parameters:
** None - Uses with data in the global "grad" record.
** Returns:
** TRUE/FALSE - Changes were made to the database.
** Sets the global "grad" record with the new data. */

int Student_Info_Changed()
## {
## int changed;    /* Changes made to data in form */
## int valid_advisor;    /* Valid advisor name ? */
## extern int *studentfrm; /* Compiled form - UNIX */
/* For VMS use 'globalref int *studentfrm;' for the compiled form */

/* Control ADDFORM to only initialize once */
   static int loadform = 0;

if (!loadform)
  {
##  message "Loading Student form . . ."
##  addform studentfrm
    loadform = 1;
   }

## display #studentfrm fill
## initialize
## (sname = grad.sname,
##  sage = grad.sage,
##  sbdate = grad.sbdate,
##  sgpa = grad.sgpa,
##   sidno = grad.sidno,
##   scomment = grad.scomment,
##   sadvisor = grad.sadvisor)

##  activate menuitem "Write"
##  {

/*
  ** If changes were made then update the database
  ** table. Only bother with the fields that are not
  ** read-only.
```

```
  */
## inquire_frs form (changed = change)

if (changed == 1)
   {
##  validate
##  message "Writing changes to database. . ."

##  getform
##  (grad.sgpa = sgpa,
##   grad.scomment = scomment,
##   grad.sadvisor = sadvisor)

/* Enforce integrity of professor name */
   valid_advisor = 0;
## retrieve (valid_advisor = 1)
##   where p.pname = grad.sadvisor

if (valid_advisor == 0)
    {
##    message "Not a valid advisor name"
##    sleep 2
##    resume field sadvisor
     }
     else
     {
##    replace s (sgpa = grad.sgpa, scomment = grad.scomment,
##     sadvisor = grad.sadvisor)
##     where s.sidno = grad.sidno
##    breakdisplay
     }
   }
## }     /* "Write" */

## activate menuitem "Quit"
## {
     /* Quit without submitting changes */
     changed = 0;
##   breakdisplay
## }     /* "Quit" */

## finalize

return (changed == 1);

## } /* Student_Info_Changed */
```

An Interactive Database Browser Using Param Statements

This application lets the user browse and update data in any table in any database. You should already have used VIFRED to create a default form based on the database table to be browsed. VIFRED builds a form whose fields have the same names and data types as the columns of the database table specified.

The program prompts the user for the name of the database, the table, and the form. In the **Get_Form_Data** procedure, it uses the **formdata** statement to find out the name, data type and length of each field on the form. It uses this information to dynamically build the elements for the **param** versions of the **retrieve**, **append**, **putform** and **getform** statements. These elements include the **param** target string, which describes the data to be processed, and the array of variable addresses, which informs the statement where to get or put the data. The type information the **formdata** statement collects includes the option of making a field nullable. If a field is nullable, the program builds a target string that specifies the use of a null indicator, and it sets the corresponding element of the array of variable addresses to point to a null indicator variable.

After the components of the **param** clause have been built, the program displays the form. If the user selects the **Browse** menu item, the program uses a **param** version of the **retrieve** statement to obtain the data. For each row, the **putform** and **redisplay** statements exhibit this data to the user. A **submenu** allows the user to get the next row or to stop browsing. When the user selects the **Insert** menu item, the program uses the **param** versions of the **getform** and **append** statements to add a new row to the database.

```
/*
** Global declarations
*/
/*
** Target string buffers for use in PARAM clauses of GETFORM,
** PUTFORM, APPEND and RETRIEVE statements. Note that the APPEND
** and PUTFORM statements have the same target string syntax.
** Therefore in this application, because the form used
** corresponds exactly to the database table, these two statements
** can use the same target string, "put_target_list".
*/
## char  put_target_list[1000] = {0};
##                             /* For APPEND and PUTFORM statements */
## char  get_target_list[1000] = {0};   /* For GETFORM statement */
## char  ret_target_list[1000] = {0};  /* For RETRIEVE statement */

# define   maxcols       127     /* DB maximum number of columns */
# define   charbufsize   3000    /* Size of "pool" of char strings */

/*
** An array of addresses of program data for use in the PARAM
** clauses. This array will be initialized by the program to point
** to variables and null indicators.
*/
## char          *var_addresses[MAXCOLS*2];
                        /* Addresses of variables and indicators */

/*
** Variables for holding data of type integer, float and
** character string.Note that to economize on memory usage,
** character data is managed as segments on one large array,
** "char_vars". Numeric variables and indicators are managed as an
** array of structures. The addresses of these data areas
** are assigned to the "var_addresses" array, according to
** the type of the field/database column.
*/
char   char_vars[CHARBUFSIZE +1];  /* Pool for character data */

struct {
```

```
          int        intv;         /* For integer data */
          double     fltv;         /* For floating-point data */
          short      indv;         /* For null indicators */
}  vars[MAXCOLS];

/*
** Procedure: main
** Purpose: Start up program and Ingres, prompting user for
**          names of form and table. Call Get_Form_Data() to obtain
**          profile of form. Then allow user to interactively
**          browse the database table and/or append new data.
*/

## main()
## {
##      char  dbname[25], formname[25], tabname[25];
##  int     inq_error;        /* Catch database and forms errors *
##  int     num_updates;      /* Catch error on database appends */
    int     want_next;            /* Browse flag */

##  forms

##  prompt ("Database name: ", dbname)
    /*
    ** Use of "-E" flag tells Ingres not to quit on start-up
    **  errors
    */
##  ingres "-E" dbname
##  inquire_ingres (inq_error = errorno)
    if (inq_error > 0)
    {
##        message "Could not start Ingres. Exiting."
##        exit
##        endforms
          exit(-1);
    }

    /* Prompt for table and form names */
##  prompt ("Table name: ", tabname)
##  range of t is tabname
##  inquire_ingres (inq_error = errorno)
    if (inq_error > 0)
    {
##        message "Nonexistent table. Exiting."
##        exit
##        endforms
          exit(-1);
    }

##   prompt ("Form name: ", formname)
##   forminit formname

     /* All forms errors are reported through INQUIRE_FRS */
##   inquire_frs frs (inq_error = errorno)
     if (inq_error > 0)
     {
##          message "Could not access form. Exiting."
##          exit
##          endforms
            exit(-1);
      }

      /*
      ** Get profile of form. Construct target lists and access
      ** variables for use in queries to browse and update data.
      */
```

```
      if (!Get_Form_Data (formname, tabname))
      {
##          message "Could not profile form. Exiting."
##          exit
##          endforms
            exit(-1);
      }

    /*
    ** Display form and interact with user, allowing browsing and
    ** appending of new data.
    */
##    display formname fill
##    initialize
##    activate menuitem "Browse"
##    {
       /*
       ** Retrieve data and display first row on form, allowing
       ** user to browse through successive rows. If data types
       ** from table are not consistent with data descriptions
       ** obtained from user's form, a retrieval error will
       ** occur. Inform user of this or other errors.
       ** Sort on first column. Note the use of "ret_varN" to
       ** indicate the column name to sort on.
       */
##       retrieve (param(ret_target_list, var_addresses))
##             sort by ret_var1
##     {
            want_next = 0;
##            putform formname (param(put_target_list, var_addresses))
##            inquire_frs frs (inq_error = errorno)
            if (inq_error > 0)
            {
##              message "Could not put data into form"
##              endretrieve
             }
            /* Display data before prompting user with submenu */
##            redisplay
##            submenu
##            activate menuitem "Next"
##            {
##              message "Next row"
                want_next = 1;
##            }
##            activate menuitem "End"
##            {
##              endretrieve
##            }
##      }    /* End of Retrieve Loop */

##     inquire_ingres (inq_error = errorno)
       if (inq_error > 0)
       {
##              message "Could not retrieve data from database"
       }
       else if (want_next == 1)
       {
            /* Retrieve loop ended because of no more rows */
##          message "No more rows"
       }
##     sleep 2

      /* Clear fields filled in submenu operations */
##       clear field all
##    }
```

```
##    activate menuitem "Insert"
##    {
##     getform formname (param(get_target_list, var_addresses))
##     inquire_frs frs (inq_error = errorno)
       if (inq_error > 0)
       {
##          clear field all
##          resume
       }
##     append to tabname (param(put_target_list, var_addresses))
##     inquire_ingres (inq_error = errorno, num_updates = rowcount)
       if (inq_error > 0 || num_updates == 0)
       {
##           message "No rows appended because of error."
       }
       else
       {
##           message "One row inserted"
       }
##     sleep 2
##    }

##    activate menuitem "Clear"
##    {
##           clear field all
##    }

##    activate menuitem "End"
##    {
##         breakdisplay
##    }

##    finalize
##    exit
##    endforms
##    }

/*
** Procedure: Get_Form_Data
** Purpose:   Get the name and data type of each field of a
**            form using the FORMDATA loop. From this information,
**            build the target strings and array of variable
**            addresses for use in the PARAM target list of
**            database and forms statements. For example, assume
**            the form has the following fields:
**
**
**            Field name    Type           Nullable?
**            ----------    ----           ---------
**            name          character      No
**            age           integer        Yes
**            salary        money          Yes
**
**            Based on this form, this procedure will construct
**            the following target string for the PARAM clause of
**            a PUTFORM statement:
**
**              "name = %c, age = %i4:%i2, salary = %f8:i2"
**
**            Note that the target strings for other statements
**            have differing syntax, depending on whether the
**            field/column name or the user variable is the
**            target of the statement.
**
**            The other element of the PARAM clause, the
**            "var_addresses" array, would be constructed by
```

```
**            this procedure as follows:
**
**               var_addresses[0] = pointer into "char_vars" array
**               var_addresses[1] = address of vars[0].intv
**               var_addresses[2] = address of vars[0].indv
**               var_addresses[3] = address of vars[1].fltv
**               var_addresses[4] = address of vars[1].indv
**
** Parameters:
**               formname
**               - Name of form to profile.
*/

## int
## Get_Form_Data(formname)
##   char  *formname;
## {
##   int     inq_error;
##   int     fld_type;          /* Data type of field */
##   char    fld_name[25];      /* Name of field */
##   int     fld_length;        /* Length of (character) field */
##   int     is_table;          /* Is field a table field? */
     char    loc_target[15];    /* Temporary target description */
     int     addr_cnt = 0;      /* Number of variable addresses */
     int     fld_cnt = 0;  /* Index to variable structures array */
     char    *char_ptr = char_vars; /* Index into character pool */
     int     ret_stat = 1;      /* Return status */

     /* Data types of fields */
#    define        date        3
#    define        money       5
#    define        int         30
#    define        float       31
#    define        char        20
#    define        vchar       21
#    define        c           32
#    define        text        37

##      formdata formname
##      {
          /* Get data information and name of each field */
##         inquire_frs field "" (fld_type=datatype, fld_name=name,
##                               fld_length = length, is_table = table)

           /* Return on errors */
##         inquire_frs frs (inq_error = errorno)
           if (inq_error > 0)
           {
              ret_stat = 0;
##            enddata
           }

           /*
           ** This application does not process table fields.
           ** However, the TABLEDATA statement is available
           ** to profile table fields.
           */
           if (is_table == 1)
           {
##              message "Table field in form"
##              sleep 2
                ret_stat = 0;
##              enddata
           }

           /* More fields than allowable columns in database? */
```

```
            if (fld_cnt >= maxcols)
            {
##            message
##             "Number of fields exceeds allowable database columns"
##                 sleep 2
                   ret_stat = 0;
##                 enddata
            }

           /* Separate target list items with commas */
            if (fld_cnt > 0)
            {
               strcat (put_target_list, ",");
               strcat (get_target_list, ",");
               strcat (ret_target_list, ",");
            }

         /* Field/col name is the target in put/append statements */
           strcat (put_target_list, fld_name);

          /*
          ** Enter data type information in target list. Point array
          ** of addresses into relevant data pool. Note that by
          ** testing the absolute value of the data type value, the
          ** program defers the question of nullable data to a later
          ** segment of the code, where it is handled in common for
          ** all types.
          ** (Recall that a negative data type indicates a nullable
          ** field.)
          */

           switch (abs(fld_type))
           {
              case int:
                  strcat (put_target_list, "= %i4");
                  strcat (get_target_list, "%i4");
                  strcat (ret_target_list, "%i4");
                  var_addresses[addr_cnt++]
                               = (char *)&vars[fld_cnt].intv;
                  break;
              case float:
              case money:
                  strcat (put_target_list, "= %f8");
                  strcat (get_target_list, "%f8");
                  strcat (ret_target_list, "%f8");
                  var_addresses[addr_cnt++] =
                            (char *)&vars[fld_cnt].fltv;
                  break;
              case c:
              case char:
              case text:
              case vchar:
              case date:
               strcat (put_target_list, "=%c");
               sprintf (loc_target, "%%c%d", fld_length);
               strcat (get_target_list, loc_target);
               strcat (ret_target_list, loc_target);
               /*
               ** Assign a segment of character buffer as space
               ** for data associated with this field. If
               ** assignment would cause overflow, give error
               ** and return.
               */
               if (char_ptr + fld_length >= &char_vars[CHARBUFSIZE])
               {
##               message "Character data fields will cause overflow"
```

```
##                  sleep 2
                    ret_stat = 0;
##                  enddata
                }
                  var_addresses[addr_cnt++] = char_ptr;
                  char_ptr += fld_length +1;
                             /* Allow room for terminator */
                  break;
                  default:
##                  message "Field has unknown data type"
                    ret_stat = 0;
##                  enddata
          }    /* End switch */

          /* If field is nullable, complete target lists
          ** and address assignments to allow for null data.
          */
          if (fld_type \ 0)
          {
              strcat (put_target_list, ":%i2" );
              strcat (get_target_list, ":%i2" );
              strcat (ret_target_list, ":%i2" );
          var_addresses[addr_cnt++] = (char *)&vars[fld_cnt].indv;
          }

          /* Ready for next structure variable */
          fld_cnt++;

          /*
          ** Field/column name is the object in getform/retrieve
          */ statements */
          strcat (get_target_list, "=");
          strcat (get_target_list, fld_name);
          strcat (ret_target_list, "=");
          strcat (ret_target_list, "t.");
          strcat (ret_target_list, fld_name);

##      }     /* End of formdata loop */

     return ret_stat;
##  }  /* Get_Form_Data */
```

Chapter 3: Embedded QUEL for COBOL

This chapter describes the use of EQUEL with the COBOL programming language.

EQUEL Statement Syntax for COBOL

This section describes the language-specific ground rules for embedding QUEL database and forms statements in a COBOL program. An EQUEL statement has the following general syntax:

*EQUEL_statement*

For information on QUEL statements, see the *QUEL Reference Guide*. For information on EQUEL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe how to use the various syntactical elements of EQUEL statements as implemented in COBOL.

Margin

There are no specified margins for EQUEL statements in COBOL. Because you must always place the two number signs (**##**) in the first two positions of the line, COBOL sequence numbers are not allowed in EQUEL lines. The rest of the statement can begin anywhere else on the line.

UNIX

The COBOL code that the preprocessor generates conforms to Micro Focus COBOL II source code format. For more details on the output format, see Precompiling, Compiling, and Linking an EQUEL Program in this chapter.

VMS

The COBOL code generated by the preprocessor conforms to COBOL source code format (ANSI or VAX COBOL terminal format, depending on whether you specify the -a flag in the preprocessor terminal line). For more details on the output format, see Precompiling, Compiling, and Linking an EQUEL Program in this chapter.

Terminator

An EQUEL/COBOL statement does not need a statement terminator. However, do use a COBOL separator period to terminate an EQUEL statement if that statement marks the end of a group of COBOL statements. For example, the separator period can appear after an EQUEL statement that indicates the end the scope of an **IF** statement as follows:

```
     IF (GIVE_MESSAGE = 1) THEN
##   MESSAGE "Continuing with processing"
##   SLEEP 2.
```

When it translates the above code into COBOL statements, the preprocessor places the COBOL separator period at the end of the last generated COBOL statement. For more details on the COBOL separator period and EQUEL statements, see Precompiling, Compiling, and Linking an EQUEL Program in this chapter.

Because variables declared to EQUEL follow the normal COBOL declaration syntax, you must terminate variable declarations in the normal way for COBOL, with a period.

Line Continuation

There are no special line continuation rules for EQUEL/COBOL. You can break an EQUEL statement between words and continue it on any number of subsequent lines. An exception to this rule is that you cannot continue a statement between two words that are reserved when they appear together, such as **declare cursor**. For a list of double keywords, see the *QUEL Reference Guide*. Start each continuation line with the ## characters. You can use blank lines between continuation lines.

If you want to continue a character-string constant across two lines, end the first line with a backslash character (\) and continue the string at the beginning of the next line, in the area which is sometimes used for sequence numbers of COBOL statements. In this case, do not place the ## characters at the beginning of the continuation lines.

For examples of string continuation, see String Literals in this chapter.

Comments

Two kinds of comments can appear in an EQUEL program: EQUEL comments and host language comments. The /* and */ characters delimit EQUEL comments and must appear on lines beginning with the ## sign.

For example:

```
##   /* Update name and salary */
##   APPEND TO EMPLOYEE (ename = EMPNAME, esal = esal*.1)
##   MESSAGE "salary updated"   /* Updates done */
```

The preprocessor strips EQUEL comments that appear on lines beginning with the ## sign out of the program. These comments do not appear in the output file.

The preprocessor treats host language comments that appear on lines that do not begin with the ## sign as host code. It passes them through to the output file unchanged. Therefore, if you want source code comments in the preprocessor output, enter them as COBOL comments.

The following restrictions only apply to EQUEL comments:

- In general, EQUEL comments can be put in EQUEL statements wherever a space can legally occur. However, comments cannot appear between two words that are reserved when they appear together, such as **declare cursor**. See the list of EQUEL reserved words in the *QUEL Reference Guide.*
- EQUEL comments cannot appear in string constants. If this occurs, the preprocessor interprets the intended comment as part of the string constant.

The following additional restrictions apply only to COBOL comments:

- COBOL comments cannot appear between component lines of EQUEL block-type statements. These include **retrieve**, **initialize**, **activate**, **unloadtable**, **formdata**, and **tabledata**, all of which have optional accompanying blocks delimited by open and close braces. Do not put COBOL comment lines between the statement and its block-opening delimiter.

  For example:

  ```
  ##  RETRIEVE (ENAME = employee.name)
  *   Illegal to put a host comment here!
  ##  {
  *   A host comment is perfectly legal here
      DISPLAY "Employee name is" ENAME
  ##  }
  ```

- COBOL comments cannot appear between the components of compound statements, in particular the **display** statement. It is illegal for a COBOL comment to appear between any two adjacent components of the **display** statement, including **display** itself and its accompanying **initialize**, **activate**, and **finalize** statements.

  For example:

  ```
  ##  DISPLAY EMPFORM
  *   illegal to put a host comment here!
  ##  INITIALIZE (empname = "FRISCO McMULLEN")
  *   Host comment illegal here!
  ##  ACTIVATE MENUITEM "Clear":
  ```

```
##  {
*   Host comment here is fine
            CLEAR FIELD ALL
##  }
*   Host comment illegal here!
##  ACTIVATE MENUITEM "End":
##  {
##  BREAKDISPLAY
##  }
*   Host comment illegal here!
##  FINALIZE
```

These restrictions are discussed on a statement-by-statement basis in the *QUEL Reference Guide.*

On the other hand, EQUEL comments are legal in the locations described in the previous paragraph, as well as wherever a host comment is legal. For example:

```
##  RETRIEVE (ENAME = employee.name)
##  /* This is an EQUEL comment, legal in this location
##     and it can span multiple lines */
##  {
       DISPLAY "Employee name" ENAME
    . . .
##  }
```

String Literals

You use double quotes to delimit string literals in EQUEL/COBOL. You can embed double quotes as part of the literal itself by doubling it. For example:

```
##  APPEND comments
##  (field1 = "a double "" quote is in this string")
```

The COBOL single quote character delimiter is also accepted by the preprocessor and is converted to a double quote.

To continue an EQUEL statement to additional lines, use the backslash (\) character at the end of the first line. Any leading spaces on the next line are considered part of the string. Therefore, the continued string should start in column 1 on the next line in the area that would be considered the Sequence Number Area on COBOL lines.

For example, the following is a legal EQUEL statement:

```
##  APPEND TO employee (empname = "Freddie \
Mac", empnum = 222)
```

Note that any string literals that are generated as output by the preprocessor will follow COBOL rules.

The Param Function

EQUEL/COBOL does not currently support **param** versions of statements. **Param** statements are supported in EQUEL/C, EQUEL/Fortran, and EQUEL/PL1.

COBOL Variables and Data Types

This section describes how to declare and use COBOL program variables in EQUEL.

Variable and Type Declarations

This section describes how to declare variables to EQUEL. It provides a general description of declaration sections and a detailed description of the declaration syntax for all data types.

EQUEL Variable Declaration Procedures

Any COBOL language variable an EQUEL statement uses must be made known to the processor so that it can determine the type of the variable. Use two number signs (##) to begin a declaration of a variable in an EQUEL/COBOL program. Begin the signs in the first column position of the line. If the variable is not used in an EQUEL statement, you do not need to use number signs, and the rules in the following sections do not apply.

Declare EQUEL/COBOL variables in the FILE or the STORAGE sections of the DATA DIVISION.

The Declare Statement

The WORKING-STORAGE SECTION for each program block must include the EQUEL statement:

```
## DECLARE
```

This statement makes the preprocessor generate a COBOL COPY statement of a file of declarations needed by the Ingres runtime system. You cannot successfully compile an EQUEL/COBOL program unless you include the **declare** statement in the WORKING-STORAGE SECTION.

Data Item Declaration Syntax

This section describes rules and restrictions for declaring EQUEL/COBOL data items. EQUEL recognizes only a subset of legal COBOL declarations.

The following template is the complete data item declaration format that EQUEL/COBOL accepts:

level-number
[*data-name* | **FILLER**]
[**REDEFINES** *data-item*]
[[**IS**] **GLOBAL**]
[[**IS**] **EXTERNAL**]
[**PICTURE** [**IS**] *pic-string*]
[[**USAGE** [**IS**]] *use-type*]
[**SIGN** *clause*]
[**SYNCHRONIZED** *clause*]
[**JUSTIFIED** *clause*]
[**BLANK** *clause*]
[**VALUE** *clause*]
[**OCCURS** *clause*] **.**

Syntax Notes:

- Data declaration clauses can be in any order, with the following two exceptions:
  - The *data-name* or **FILLER** clause, if given, must immediately follow the level number.
  - The **REDEFINES** clause, if given, must immediately follow the *data-item* or **FILLER** clause.
- The *level-number* can range from 01 to 49. Level number 77 (for noncontiguous data items) is also valid and the preprocessor regards it as identical to level 01. The EQUEL/COBOL preprocessor does not support levels 66 (which identifies **RENAMES** items) and 88 (which associates condition names with values).

  Follow the COBOL rules for specifying the organization of data when you assign level numbers to your EQUEL data items. Like the COBOL compiler, the preprocessor recognizes that a data item belongs to a record or group if its level number is greater than the record or group level number.
- The *data-name* must begin with an alphabetic character or an underscore, which can be followed by alphanumeric characters, hyphens, and underscores. The word **FILLER** may appear in place of *data-name*; however, you cannot explicitly reference a **FILLER** item in an EQUEL statement. If the *data-name* or **FILLER** clause is omitted, **FILLER** is the default.

- The preprocessor accepts but does not use the **REDEFINES**, **GLOBAL**, **EXTERNAL**, **SIGN**, **SYNCHRONIZED**, **JUSTIFIED**, **BLANK**, and **VALUE** clauses. Consequently, illegal use of these clauses goes undetected at preprocess time but generates COBOL errors later at compile time. For example, the preprocessor does not check that a **GLOBAL** clause appears only on an 01 level item, nor that a **SIGN** clause appears only on a numeric item.
- The preprocessor expects a **PICTURE** clause on the **COMP**, **COMP-3**, **COMP-5** (UNIX only) and **DISPLAY** *use-types*.

  Do not use a **PICTURE** clause on **INDEX** *use-types* and on the UNIX **COMP-1 and COMP-2** *use-types.*

  Although the preprocessor recognizes all the valid COBOL **PICTURE** symbols, it only makes use of the type and size information needed for runtime support. It does not, for instance, complain about certain illegal combinations of editing symbols in picture strings. EQUEL accepts **PIC** as an abbreviation for **PICTURE**. You must specify the picture string on the same line as the keyword **PICTURE**.
- For a description of the valid *use-types* for the **USAGE** clause and their interaction with picture strings, see Data Types in this chapter.
- The preprocessor accepts the **OCCURS** clause for all data items in the level range 02 through 49. The preprocessor does not use the information in the **OCCURS** clause, except to note that the item described is an array. If you use an **OCCURS** clause on level 01, the preprocessor issues an error but generates correct code so that you can compile and link the program.

Reserved Words in Declarations

You cannot declare types or variables with the same name as EQUEL keywords. You can only use them in quoted string constants. All EQUEL keywords are reserved. In addition to EQUEL keywords, the following EQUEL/COBOL keywords are reserved and cannot be used except in quoted string constants.

ASCENDING	**DECLARE**	**PACKED_DECIMAL**
BLANK	**DEPENDING**	**PIC**
BY*	**DESCENDING**	**PICTURE**
CHARACTER	**DISPLAY***	**POINTER**
COMP-1	**EXTERNAL**	**REDEFINES**
COMP-2	**FILLER**	**REFERENCE**

COMP-3	**GLOBAL***	**SEPARATE**
COMP-4	**IN***	**SIGN**
COMP-5	**INDEX***	**SYNC**
COMP-6	**INDEXED**	**SYNCHRONIZED**
COMP	**IS***	**TIMES**
COMPUTATIONAL-1	**JUST**	**TO**
COMPUTATIONAL-2	**JUSTIFIED**	**TRAILING**
COMPUTATIONAL-3	**KEY***	**USAGE**
COMPUTATIONAL-4	**LEADING**	**VALUE**
COMPUTATIONAL-5	**OCCURS**	**WHEN**
COMPUTATIONAL-6	**OF***	**ZERO**
COMPUTATIONAL	**ON***	

The EQUEL preprocessor does not distinguish between uppercase and lowercase in keywords. When it generates COBOL code, it converts any lowercase letters in keywords to uppercase. This rule is true only for keywords. The preprocessor does distinguish between case in program-defined types and variable names.

Variable and type names must be legal COBOL identifiers beginning with an alphabetic character or an underscore.

Data Types

EQUEL/COBOL supports a subset of the COBOL data types. The following table maps the COBOL data types to their corresponding Ingres types. Note that the COBOL data type is determined by its category, picture and usage.

COBOL Type			Ingres Type
Category	**PICTURE**	**USAGE**	
ALPHABETIC	any	**DISPLAY**	**character**
ALPHANUMERIC	any	**DISPLAY**	**character**
ALPHANUMERIC EDITED	any	**DISPLAY**	**character**
NUMERIC	**9(***p***)** where *p* <= 10	**COMP, DISPLAY**	**integer**
NUMERIC	**9(***p***)V9(***s***)** where *p*+*s* <= 9	**COMP, DISPLAY**	**float**

COBOL Type			Ingres Type
Category	**PICTURE**	**USAGE**	
NUMERIC	**9(*p*)** where *p* <= 10	**COMP-3**	**integer**
NUMERIC		**INDEX**	**integer**
NUMERIC EDITED	any	**DISPLAY**	**integer, float**
NUMERIC		**COMP-1**	**float**
NUMERIC		**COMP-2**	**float**

VMS

COMP is an abbreviation for **COMPUTATIONAL**. You can use either form. Note that POINTER data items are not supported. The following sections describe the various data categories and the manner in which EQUEL interacts with them.

The Numeric Data Category - UNIX

EQUEL/COBOL accepts the following declarations of numeric variables:

level-number data-name **PIC** [**IS**] *pic-string* [**USAGE** [**IS**]]
COMP|**COMP-3**|**COMP-5** |**DISPLAY.**
level-number data-name [**USAGE** [**IS**]] **INDEX.**

Syntax Notes:

- Use the symbol "S" on numeric picture strings to indicate the presence of an operational sign.
- The picture string (*pic-string*) of a **COMP**, **COMP-3, COMP-5** data item can contain only the symbols "9", "S", and "V" in addition to the parenthesized length.
- In order to interact with Ingres integer-valued objects, the picture string of a **COMP**, **COMP-3** or **DISPLAY or COMP-5** item must describe a maximum of 10 digit positions with no scaling.
- Do not use a picture string for **INDEX** data items. While the preprocessor ignores such a picture string, the compiler does not allow it.

You can use any data items in the numeric category to assign and receive Ingres numeric data in database tables and forms. However, you can only use non-scaled **COMP**, **COMP-3**, **COMP-5** and **DISPLAY** items of 10 digit positions or less to specify simple numeric objects, such as table field row numbers. Generally, try to use **COMP** data items with no scaling to interact with Ingres integer-valued objects, since the internal format of **COMP** data is compatible with Ingres integer data.

Ingres effects the necessary conversions between all numeric data types, so the use of **DISPLAY** and **COMP-3** scaled data items is allowed. For more information on type conversion, see Data Type Conversion in this chapter.

The following example contains numeric data categories:

```
##  01 QUAD-INTVAR       PIC S9(10) USAGE COMP.
##  01 LONG-INTVAR       PIC S9(9) USAGE COMP.
##  01 SHORT-INTVAR      PIC S9(4) USAGE COMP.
##  01 DISPLAY-VAR       PIC S9(10) USAGE DISPLAY.
##  01 PACKED-VAR        PIC S9(12)V9(4) USAGE COMP-3.
```

Numeric Data Items with Usage COMP-5 - UNIX

Ingres supports data items declared with **USAGE COMP-5**. When you specify this clause, the data item is stored in the same machine storage format as the native host processor rather than in the byte-wise Micro Focus storage format. Of course, sometimes the two storage formats are identical. Since the Ingres runtime system that is linked into your COBOL runtime support module (RTS) is written in C, it is important that Ingres interact with native data types rather than Micro Focus data types. Consequently, many of your normal **USAGE COMP** data items are transferred (using COBOL **MOVE** statements) into internally declared Ingres **USAGE COMP-5** data items. Data items declared with this **USAGE** will cause a compiler informational message (209-I) to occur.

The Numeric Data Category - VMS

EQUEL/COBOL accepts the following declarations of numeric variables:

level-number data-name **PIC** [**IS**] *pic-string* [**USAGE** [**IS**]]
COMP|**COMP-3**|**COMP-5** |**DISPLAY.**
level-number data-name [**USAGE** [**IS**] **COMP-1|COMP-2| INDEX.**

Syntax Notes:

- Use the symbol "S" on numeric picture strings to indicate the presence of an operational sign.
- The picture string (*pic-string*) of a **COMP**, **COMP-3** data item can contain only the symbols "9", "S", and "V" in addition to the parenthesized length.
- In order to interact with Ingres integer-valued objects, the picture string of a **COMP**, **COMP-3**, or **DISPLAY** item must describe a maximum of 10 digit positions with no scaling.
- Do not use a picture string for **INDEX**, **COMP-1,** or **COMP-2** data items. While the preprocessor ignores such a picture string, the compiler does not allow it.

You can use any data items in the numeric category to assign and receive Ingres numeric data in database tables and forms. However, you can only use non-scaled **COMP**, **COMP-3**, and **DISPLAY** items of 10 digit positions or less to specify simple numeric objects, such as table field row numbers. Generally, try to use **COMP** data items with no scaling to interact with Ingres integer-valued objects, since the internal format of **COMP** data is compatible with Ingres integer data.

Similarly, COMP-1 and COMP-2 data items are compatible with Ingres floating-point data.

Ingres effects the necessary conversions between all numeric data types, so the use of **DISPLAY** and **COMP-3** scaled data items is allowed. For more information on type conversion, see Data Type Conversion in this chapter.

The following example contains numeric data categories:

```
##  01 QUAD-INTVAR        PIC S9(10) USAGE COMP.
##  01 LONG-INTVAR        PIC S9(9) USAGE COMP.
##  01 SHORT-INTVAR       PIC S9(4) USAGE COMP.
##  01 DISPLAY-VAR        PIC S9(10) USAGE DISPLAY.
##  01 SING-FLOATVAR     USAGE COMP-1.
##  01 DOUB-FLOATVAR     USAGE COMP-2.
##  01 PACKED-VAR         PIC S9(12)V9(4) USAGE COMP-3.
```

The Numeric Edited Data Category

The syntax for a declaration of numeric edited data is:

level-number data-name **PIC** [**IS**] *pic-string* [[**USAGE** [**IS**]] **DISPLAY**]**.**

Syntax Notes:

- The *pic-string* can be any legal COBOL picture string for numeric edited data. The preprocessor notes only the type, scale, and size of the data item.
- In order to interact with Ingres integer-valued objects, the picture string must describe a maximum of 10 digit positions with no scaling.

While you can use numeric edited data items to assign data to, and receive data from, Ingres database tables and forms, be prepared for some loss of precision for numeric edited data items with scaling. The runtime interface communicates using integer (**COMP**) or packed **(COMP-3)** variables for UNIX or float **(COMP-2)** variables for VMS.

In moving from these variables into your program's edited data items, truncation can occur due to **MOVE** statement rules and the COBOL standard alignment rules. For more information on type conversion, see Data Type Conversion in this chapter.

The following example illustrates the numeric edited data category:

```
##  01 DAILY-SALES          PIC $$$,$$9DB USAGE DISPLAY.
##  01 GROWTH-PERCENT       PIC ZZZ.9(3) USAGE DISPLAY.
```

The Alphabetic, Alphanumeric, and Alphanumeric Edited Categories

EQUEL/COBOL accepts data declarations in the alphabetic, alphanumeric, and alphanumeric edited categories. The syntax for declaring data items in those categories is:

level-number data-name **PIC** [**IS**] *pic-string*
[[**USAGE** [**IS**]] **DISPLAY**].

Syntax Note:

- The *pic-string* can be any legal COBOL picture string for the alphabetic, alphanumeric and alphanumeric edited classes. The preprocessor notes only the length of the data item and that the data item is in the alphanumeric class.

You can use alphabetic, alphanumeric, and alphanumeric edited data items with any Ingres object of character (**c**, **char**, **text** or **varchar**) type. You can also use them to replace names of objects such as forms, fields, tables and columns. However, when a value is transferred into a data item from a Ingres object it is copied directly into the variable storage area without regard to the COBOL special insertion rules. When data in the database is in a different format from the alphanumeric edited picture, you must provide an extra variable to receive the data. You can then **MOVE** the data into the alphanumeric edited variable. However, if data in the database is in the same format as the alphanumeric edited picture (which would be the case, for example, if you had inserted data using the same variable you are retrieving into), you can assign the data directly into the edited data item, without any need for the extra variable. For more information on type conversion, see Data Type Conversion in this chapter.

The following example illustrates the syntax for these categories:

```
##  01 ENAME          PIC X(20).
##  01 EMP-CODE       PIC xx/99/00.
```

Declaring Records

EQUEL/COBOL accepts COBOL record and group declarations. The following syntax declares a record:

01 *data-name***.**
*record-item***.**
{*record-item***.**}

where *record-item* is a group item:

*level-number data-name***.**
*record-item***.**
{*record-item***.**}

or an elementary item:

*level-number data-name elementary-item-description***.**

Syntax Notes:

- The record must have a level number of 01. Thereafter, the level numbers of *record-items* can be 02 through 49. The preprocessor applies the same rules as the COBOL compiler in using the level numbers to order the groups and elementary items in a record definition into a hierarchical structure.
- If you do not specify the *elementary-item-description* for a record item, the record item is assumed to be a group item.
- The *elementary-item-description* can consist of any of the attributes described for data declarations (see Data Item Declaration Syntax in this chapter). The preprocessor does not confirm that the different clauses are acceptable for record items.
- The **OCCURS** clause, denoting a COBOL table, may appear on any record item.
- Only record-items that EQUEL statements reference need to be declared to EQUEL. The following example declares a COBOL record with several "filler" record-items that are not declared to EQUEL:

```
##   01   PERSON-REC.
##        02   NAME.
##             03    FIRST-NAME      PIC X(10).
                   03    FILLER          PIC X.
##             03    LAST-NAME       PIC X(15).
##        02   STREET-ADDRESS.
##             03    ST-NUMBER       PIC 99999 DISPLAY.
                   03    FILLER          PIC X.
##             03    STREET          PIC X(30).
##        02   TOWN-STATE.
##             03    TOWN            PIC X(20).
                   03    FILLER          PIC X.
##             03    STATE           PIC X(3).
                   03    FILLER          PIC X.
##             03    ZIP             PIC 99999 DISPLAY.
```

Indicator Data Items

An *indicator data item* is a 2-byte integer numeric data item. There are three possible ways to use these in an application:

- In a statement that retrieves data from Ingres, you can use an indicator data item to determine if its associated host variable was assigned a null value.
- In a statement that sets data to Ingres, you can use an indicator data item to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator data item as a check that the associated host variable is large enough to hold the full length of the returned character string.

An indicator declaration must have the following syntax:

level-number indicator-name **PIC** [**IS**] **S9(***p*) [**USAGE** [**IS**]] **COMP.**

where *p* is less than or equal to 4.

The following is an example of an indicator declaration:

```
##  01 IND-VAR
    PIC S9(2) USAGE COMP.
```

Compiling and Declaring External Compiled Forms - UNIX

You can precompile your forms in the Visual Forms Editor (VIFRED). This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in C. VIFRED prompts you for the name of the file. Once the C file is created, you can use the following command to compile it into a linkable object module:

cc -c *filename*.c

This command produces an object file containing a global symbol with the same name as your form. Before the EQUEL/FORMS statement **addform** can refer to this global object, you must use the following syntax to declare it to EQUEL:

01 *formname* [**IS**] **EXTERNAL PIC S9(9)** [**USAGE** [**IS**]] **COMP-5.**

Some platforms do not support the above syntax. If **EXTERNAL** data items cannot be referenced in your COBOL program, use an alternative procedure. For an alternate procedure, see Including External Compiled Forms in the RTS in this chapter.

Syntax Notes:

- The *formname* is the actual name of the form. VIFRED gives this name to the global object. The *formname* is used to refer to the form in EQUEL/FORMS statements *after* the form has been made known to the FRS using the **addform** statement.
- The **EXTERNAL** clause causes the linker to associate the *formname* data item with the external *formname* symbol.

The following example shows a typical form declaration and illustrates the difference between using the form's global object definition and the form's name. (Currently, this example does not work on all Micro Focus platforms.)

```
DATA DIVISION.
WORKING-STORAGE SECTION.

##   01  empform IS EXTERNAL PIC S9(9) USAGE COMP-5.

*    Other data declarations.

PROCEDURE DIVISION.

*    Program initialization.

*    Making the form known to the FRS via the global
*    form object.
##     ADDFORM empform.

*    Displaying the form via the name of the form.
##     DISPLAY #empform

*    The program continues.
```

For information on using external compiled forms with your EQUEL program, see Including External Compiled Forms in the RTS in this chapter.

Assembling and Declaring External Compiled Forms -VMS

You can pre-compile your forms in VIFRED. This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. After the MACRO file is created, you can use the VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the EQUEL/FORMS statement **addform** can refer to this global object, you must declare it to EQUEL, with the following syntax:

01 *formid* **PIC S9(9)** [**USAGE** [**IS**]] **COMP VALUE** [**IS**] **EXTERNAL** *formname***.**

Syntax Notes:

- The *formid* is a COBOL data item. It is used with the **addform** statement to declare the form to the Forms Runtime System (FRS).
- The *formname* is the actual name of the form. VIFRED gives this name to the global object. The *formname* is used to refer to the form in EQUEL/FORMS statements *after* the form is made known to the FRS with the **addform** statement.
- The **EXTERNAL** clause causes the VAX linker to associate the *formid* data item with the external *formname* symbol.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition (the *formid*) and the form's name (the *formname*).

```
DATA DIVISION.
WORKING-STORAGE SECTION.

##  01  EMPFORM-ID PIC S9(9) USAGE COMP VALUE IS EXTERNAL
##      empform.

* Other data declarations.

PROCEDURE DIVISION.

Program initialization.

*     Making the form known to the FRS via the global form
*     object.
##      ADDFORM EMPFORM-ID.

*     Displaying the form via the name of the form.
##    DISPLAY empform

*  The program continues.
```

For information on linking your EQUEL program with external compiled forms, see Assembling and Declaring External Compiled Forms -VMS in this chapter.

Concluding Examples

The following UNIX and VMS examples demonstrate some simple EQUEL declarations.

UNIX

```
*Data item to hold database name.
##   01 DBNAME       PIC X(9) VALUE IS "Personnel".

* Scaled data
##   01 SALARY       PIC S9(8)V9(2) USAGE COMP.
##   01 MONEY        PIC S999V99 USAGE COMP-3.

* Array of numerics
##   01 NUMS.
##       02 NUM-ARR            PIC S99 OCCURS 10 TIMES.

* Record of a full name and a redefinition of its parts.
##   01 NAME-REC.
##       02 FULL-NAME          PIC X(20).
##       02 NAME-PARTS REDEFINES FULL-NAME.
##          03 FIRST-NAME     PIC X(8).
##          03 MIDDLE-INIT    PIC X(2).
##          03 LAST-NAME      PIC X(10).

* Record for fetching and displaying.
#  # 01 OUT-REC.
##       02 FILLER   PIC X(15) VALUE "Value fetched: ".
##       02 FROM-DB  PIC S9(4) USAGE DISPLAY.

* Miscellaneous attributes (some declaration clauses are
* ignored by preprocessor)
##   01 SALES-TOT         PIC S9(6)V99 SIGN IS TRAILING.

##   01 SYNC-REC.
##       02 NUM1          PIC S99 USAGE COMP SYNCHRONIZED.
##       02 FILLER        PIC X VALUE SPACES.
##       02 NUM2          PIC S99 USAGE COMP SYNCHRONIZED.

##   01 RIGHT-ALIGN       PIC X(30) JUSTIFIED RIGHT.

##   01 NUM-OUT  PIC S99V99 USAGE DISPLAY BLANK WHEN ZERO.

##      DECLARE.
```

VMS

```
* Data item to hold database name.
##   01 DBNAME       PIC X(9) VALUE IS "Personnel".

* Scaled data
##   01 SALARY       USAGE COMP-1.
##   01 MONEY        PIC S999V99 USAGE COMP-3.

* Array of numerics
##   01 NUMS.
##       02 NUM-ARR         PIC S99 OCCURS 10 TIMES.

* Record of a full name and a redefinition of its parts.
##   01 NAME-REC.
##       02 FULL-NAME           PIC X(20).
##       02 NAME-PARTS REDEFINES FULL-NAME.
##           03 FIRST-NAME    PIC X(8).
```

```
##           03 MIDDLE-INIT   PIC X(2).
##           03 LAST-NAME     PIC X(10).

* Record for fetching and displaying.
##   01 OUT-REC.
##       02 FILLER    PIC X(15) VALUE "Value fetched: ".
##       02 FROM-DB   PIC S9(4) USAGE DISPLAY.

* Miscellaneous attributes (ignored by preprocessor)
##   01 SALES-TOT        PIC S9(6)V99 SIGN IS TRAILING.

##   01 SYNC-REC.
##       02 NUM1       PIC S99 USAGE COMP SYNCHRONIZED.
##       02 FILLER     PIC X VALUE SPACES.
##       02 NUM2       USAGE COMP-2 SYNCHRONIZED.

##   01 RIGHT-ALIGN    PIC X(30) JUSTIFIED RIGHT.

##   01 NUM-OUT  PIC S99V99 USAGE DISPLAY BLANK WHEN ZERO.
##      DECLARE.
```

The Scope of Variables

You can reference all variables declared to EQUEL. The preprocessor accepts them from the point of declaration to the end of the file. This is not true for the COBOL compiler, which generally allows references to only those variables declared in the current program units. Because the preprocessor does not terminate the scope of a variable in the same way the COBOL compiler does, do not redeclare variables of the same name to the preprocessor in a single file even where the variables are declared in separately compiled program units. If two programs in a single file each use variables of the same name and type in EQUEL statements, you must declare only the first with the ## signal.

Variable Usage

COBOL variables (that is, data items) declared to EQUEL can substitute for most elements of EQUEL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. The generic uses of host language variables in EQUEL statements are described the *QUEL Reference Guide*. The following discussion covers only the usage issues particular to COBOL language variable types.

The following **retrieve** statement uses the variables "NAMEVAR" and "NUMVAR" to receive data, and the variable "IDNO" as an expression in the **where** clause:

```
##   RETRIEVE (NAMEVAR = employee.empname,
##         NUMVAR = employee.empnum) WHERE
##         employee.empnum = IDNO
```

To distinguish the minus sign used as a subtraction operator in an EQUEL statement from the hyphen used as a character in a data item name, you must delimit the minus sign by blanks. For example, the statement:

```
##   APPEND TO employee (ename="Jones", eno=ENO-2)
```

indicates that the data item "ENO-2" is to be appended to column "eno". To append a value two less than the value in the data item "ENO" you must instead use the following statement:

```
##   APPEND TO employee (ename="Jones", eno=ENO - 2)
```

Note the spaces surrounding the minus sign.

Elementary Data Items

The following syntax refers to a simple scalar-valued data item (numeric, alphanumeric, or alphabetic):

simplename

The following program fragment demonstrates a typical error handling paragraph. The data items "BUFFER" and "SECONDS" are scalar-valued variables.

```
DATA DIVISION.
WORKING-STORAGE SECTION.

##   01 SECONDS       PIC S9(4) USAGE COMP.
##   01 BUFFER        PIC X(100).

##  DECLARE.

* Program code

ERROR-HANDLE.
##    MESSAGE BUFFER.
##    SLEEP SECONDS.

* More error code.
```

COBOL Tables

The following syntax refers to a COBOL array or table:

*tablename***(***subscript*{**,***subscript*}**)**

Syntax Notes:

- You must subscript the *tablename* because only elementary data items are legal EQUEL values.
- When you declare a COBOL table, the preprocessor notes from the **OCCURS** clause that it is a table and not some other data item. When you later reference the table, the preprocessor confirms that a subscript is present but does not check the legality of the subscript inside the parentheses. Consequently, you must ensure that the subscript is legal and that the correct number of subscripts are used.

In the following example, the variable "SUB1" is used as a subscript and does not need to be declared to EQUEL declaration section, because the preprocessor ignores it.

```
DATA DIVISION.
WORKING-STORAGE SECTION.

##   01  FORMNAMES.
##       02   FORM-TABLE     PIC X(8) OCCURS 3 TIMES.

##   01  SUB1           PIC S9(4) USAGE COMP VALUE ZEROES.
##   DECLARE.

PROCEDURE DIVISION.
BEGIN.

* Program code

     PERFORM VARYING SUB1 FROM 1 BY 1
           UNTIL SUB1 > 3

##         FORMINIT FORM-TABLE(SUB1).

     END-PERFORM.

* More program code.
```

Record Data Items

You cannot use a record data item (also referred to as a *structure variable*) as a single entity in an EQUEL statement. Only elementary data items can communicate with Ingres objects and data.

EQUEL and COBOL use the same syntax to refer to an elementary record item:

elementary-item-name **IN** | **OF**{ *groupname* **IN** | **OF**} *recordname*

Syntax Notes:

- The *item* in the above reference must be a scalar value (numeric, alphanumeric, or alphabetic). You can use any combination of tables and records, but the last referenced item must be a scalar value. Thus, the following references are all legal:

```
* Element of a record
SAL IN EMPLOYEE
SAL OF EMPLOYEE

* Element of a record as an item of a table
NAME IN PERSON(3)

* Deeply nested element
ELEMENTARY-ITEM OF GROUP3 OF GROUP2 OF REC
```

- The qualification of an elementary item in a record can be elliptical; that is, you do not need to specify all the names in the hierarchy in order to reference the item. You must not, however, use an ambiguous reference that does not clearly qualify an item. For example, assume the following declaration:

```
##   01   PERSON.
##        02   NAME    PIC X(30).
##        02   AGE     PIC S9(4) USAGE COMP.
##        02   ADDR    PIC X(50).
```

  If you reference the variable "NAME", the preprocessor assumes the elementary item "NAME IN PERSON" is being referred to. However, if there also was the declaration:

```
##   01   CHILD.
##        02   NAME    PIC X(30).
##        02   PARENT  PIC X(30).
```

  then the reference to "NAME" is ambiguous, because it can refer to either "NAME IN PERSON" or "NAME IN CHILD."

- Subscripts, if present, must qualify the data item declared with the **OCCURS** clause.

The following example uses the record "EMPREC" that contains the elementary data items "ENO", "ENAME," AGE," "JOB," "SALARY," and "DEPT". Assume "EMPREC" was declared to EQUEL in the file "employee.dcl".

```
DATA DIVISION.
WORKING-STORAGE SECTION.
* See above for description.
##  EXEC SQL INCLUDE "employee.dcl".
##  DECLARE.
PROCEDURE DIVISION.
* Program Code
##    PUTFORM empform
##      (eno = ENO IN EMPREC, ename = ENAME IN EMPREC,
##       age = AGE IN EMPREC, job = JOB IN EMPREC,
##       sal = SAL IN EMPREC, dept = DEPT IN EMPREC)
```

Note that you can write the **putform** statement without the "EMPREC" qualifications, assuming there were no ambiguous references to the item names:

```
##    PUTFORM empform
##      (eno = ENO, ename = ENAME, age = AGE,
##       job = JOB, sal = SAL, dept = DEPT)
```

Using Indicator Data Items

The syntax for referring to an *indicator* data item is the same as for an elementary data item, except that an indicator variable is always associated with another COBOL data item:

data_item:indicator_item

Syntax Note:

- The indicator data item must be a 2-byte integer numeric elementary data item. For example:

```
##  01 IND-1        PIC S9(4) USAGE COMP.
##  01 IND-TABLE.
##     02 IND-2     PIC S9(4) USAGE COMP OCCURS 5 TIMES.
##  01 NUMVAR       PIC S9(9) USAGE COMP.
##  01 EMPNAMES.
##     02 ENAME     PIC X(30) OCCURS 5 TIMES.
##  01 SUB1         PIC S9(4) USAGE COMP VALUE ZEROES.
##  APPEND TO employee (empnum=NUMVAR:IND-1)
##  RETRIEVE (ENAME(SUB1): IND-2(SUB1)=
##  employee.empname)
##  {
        program code
        ADD 1 TO SUB1
##  }
```

Data Type Conversion

A COBOL data item must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into COBOL numeric and numeric edited items, and Ingres character values can be set by and retrieved into COBOL character data items, that is, alphabetic, alphanumeric, and alphanumeric edited items.

Data type conversion occurs automatically for different numeric types, such as from floating-point Ingres database column values into integer COBOL variables, and for character strings, such as from varying-length Ingres character fields into fixed-length COBOL character string buffers.

Ingres does *not* automatically convert between numeric and character types, such as from Ingres integer fields into COBOL alphanumeric data items. You must use the Ingres type conversion functions, the Ingres **ascii** function, or the COBOL **STRING** statement to effect such conversion.

The following table shows the default type compatibility for each Ingres data type. Note that some COBOL types are omitted from the table because they do not exactly match a Ingres type. Use of those types necessitates some runtime conversion, which may possibly result in some loss of precision.

UNIX

There is no exact match for float, so use COMP-3.

Ingres Types and Corresponding COBOL Data Types

Ingres Type	UNIX COBOL Type	VMS COBOL Type
cN	**PIC X(*N*)**	**PIC X(*N*)**
text(*N*)	**PIC X(*N*)**	**PIC X(*N*)**
char(*N*)	**PIC X(*N*).**	**PIC X(*N*).**
varchar(*N*)	**PIC X(*N*).**	**PIC X(*N*).**
i1	**PIC S9(2) USAGE COMP.**	**PIC S9(2) USAGE COMP.**
i2	**PIC S9(4) USAGE COMP.**	**PIC S9(4) USAGE COMP.**
i4	**PIC S9(9) USAGE COMP.**	**PIC S9(9) USAGE COMP.**
f4	**PIC S9(10)V9(8) USAGE COMP-3.**	**USAGE COMP-1.**
f8	**PIC S9(10)V9(8) USAGE COMP-3.**	**USAGE COMP-2**
date	**PIC X(25).**	**PIC X(25).**
money	**PIC S9(10)V9(8) USAGE COMP-3.**	**USAGE COMP-2.**
decimal	**PIC S9(P-S)V(S) USAGE COMP-3.**	**PICS9(P-S)V(S) USAGE COMP-3.**

Note that Ingres stores **decimal** as signed. Thus, use a signed decimal variable if it interacts with a Ingres **decimal** type. Also, Ingres allows a maximum precision of 31 while COBOL allows only 18.

Decimal Type Conversion

A Ingres decimal value that will not fit into a COBOL variable will either be truncated if there is loss of scale or cause a runtime error if loss of significant digits.

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric COBOL data items. It follows the standard COBOL type conversion rules. For example, if you assign the value in a scaled **COMP-3** data item for UNIX or a **COMP-1** data item for VMS to an integer-valued field in a form, the digits after the decimal point of the data item's value are truncated. Runtime errors are generated for overflow on conversion.

The preprocessor generates COBOL **MOVE** statements to convert various COBOL data types. These can again be converted at runtime based on the final value being set or retrieved. Note that the standard COBOL data conversion rules hold for all these generated **MOVE** statements, with a potential loss of precision.

Floats are coerced to decimal types by Ingres at runtime. The preprocessor uses temporary data items when moving values between numeric **DISPLAY** data items and Ingres objects. Depending on the **PICTURE** clause of the **DISPLAY** item, these temporary data items are either:

- **COMP-3** or 4-byte **COMP-5** for UNIX

or

- **COMP-2** or 4-byte **COMP** for VMS

Numeric DISPLAY Items and Temporary Data Items

Numeric DISPLAY Item's Picture	Temporary Item's Data Type - UNIX	Temporary Item's Data Type - VMS
With scaling	**PIC S9(9)V9(9) USAGE COMP-3**	**COMP-2**
With > 10 numeric digits	**PIC S9(9)V9(9) USAGE COMP-3**	**COMP-2**
No scaling and 10 numeric digits	4-byte **COMP-5**	4-byte **COMP**

COMP-3 items used to set or receive Ingres values also require some runtime conversion. This is not true if you are setting or receiving decimal data. This is true for Micro Focus COBOL when float values are received into **COMP-3.**

The preprocessor also generates code to use a temporary data item when Ingres data is to interact with a COBOL unscaled **COMP** data item whose picture string is exactly 10.

UNIX

Because a COBOL non-scaled numeric item whose picture contains 10 or fewer digits is regarded as compatible with the Ingres **integer** type, EQUEL/COBOL assigns such data to a temporary COBOL 4-byte **COMP-5** data item to allow it to interact with Ingres **integer** data. Note that the range of the Ingres **i4** type does not include *all* 10-digit numbers. If you have 10-digit numeric data outside the Ingres range, you should use a **COMP-3** data item and choose the Ingres **float** type.

You can use only **COMP** data items or items that get assigned to temporary 4-byte **COMP-5** data items (as described above) to set the values of Ingres integer objects, such as table field row numbers. You can, however, use any numeric data items to set and retrieve numeric values in Ingres database tables or forms.

The Ingres **money** type is represented as a **COMP-3** data item. 

VMS

A COBOL non-scaled numeric item whose picture contains 10 or fewer digits is regarded as compatible with the Ingres **integer** type. However, the VAX standard data type for an unscaled 10-digit COMP item is a quadword (8 bytes). Therefore, EQUEL/COBOL assigns such data to a temporary COBOL 4-byte COMP data item to allow it to interact with Ingres **integer** data. Note that the range of the Ingres **i4** type does not include *all* 10-digit numbers. If you have 10-digit numeric data outside the Ingres range you should use a COMP-1 or COMP-2 data item and choose the Ingres **float** type.

You can use only **COMP** data items or items that get assigned to temporary 4-byte **COMP** data items (as described above) to set the values of Ingres integer objects, such as table field row numbers. You can, however, use any numeric data items to set and retrieve numeric values in Ingres database tables or forms.

The Ingres **money** type is represented as **COMP-2**, an 8-byte floating-point value. 

Runtime Character Conversion

Automatic conversion occurs between Ingres character string values and COBOL character variables (alphabetic, alphanumeric, and alphanumeric edited data items). There are four string-valued Ingres objects that can interact with character variables:

- Ingres names, such as form and column names
- Database columns of type **c** or **char**
- Database columns of type **text** or **varchar**
- Form fields of type **c**

Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of COBOL character variables used to represent Ingres names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string constants "empform " and "empform" refer to the same form and "employees " and "employees" refer to the same database table.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **c** or **char**, a database column of type **text** or **varchar**, or a character-type form field. Ingres pads columns of type **c** and **char** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **text**, or **varchar** in form fields.

Second, the COBOL convention is to blank-pad fixed-length character strings. For example, the character string "abc" may be stored in a COBOL PIC X(5) data item as the string "abc" followed by two blanks.

When character data is retrieved from a database column or form field into a COBOL character variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You must always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data. You should note that, when a value is transferred into a data item from a Ingres object, it is copied directly into the variable storage area without regard to the COBOL special insertion rules.

When inserting character data into an Ingres database column or form field from a COBOL variable, note the following conventions:

- When data is inserted from a COBOL variable into a database column of type **c** or **char** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a COBOL variable into a database column of type **text** or **varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **text** or **varchar** column. For example, when a string "abc" stored in a COBOL PIC X(5) data item as "abc " (refer to above) is inserted into the **text** or **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the Ingres **notrim** function. It has the following syntax:

  notrim(*charvar*)

  where *charvar* is a character variable. An example that demonstrates this feature follows this section. If the **text** or **varchar** column is shorter than the variable, the data is truncated to the length of the column.

- When data is inserted from a COBOL variable into a **c** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

When comparing character data in a Ingres database column with character data in a COBOL variable, note the following convention:

- When comparing data in a **c**, **character**, or **varchar** database column with data in a character variable, all trailing blank are ignored. Initial and embedded blanks are significant in **character**, **text**, and **varchar**; they are ignored in **c**.

Caution! As previously described, the conversion of character string data between Ingres objects and COBOL variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion.

The Ingres **date** data type is represented as a 25-byte character string: PIC X(25).

The program fragment in the following examples demonstrates the **notrim** function and the truncation rules explained above:

UNIX

```
DATA DIVISION.
WORKING-STORAGE SECTION
##  01 ROW        PIC S9(4) USAGE COMP.
##  01 DATA       PIC X(7).

##  DECLARE.

PROCEDURE DIVISION.
BEGIN.
##  MOVE "abc " TO DATA.

*   Set up the table for testing
##  CREATE texttype (#row = i2, #data = text(10))

*   The first APPEND adds the string "abc"  (blanks
*   truncated)
##  APPEND TO texttype (#row = 1, #data = data)

*   The second APPEND adds the string "abc ",
*   with 4 trailing   blanks
##  APPEND TO texttype (#row = 2, #data = NOTRIM(data))

*   The RETRIEVE will get the second row because
*   the NOTRIM   function in the previous APPEND
*   caused trailing blanks to be inserted as data.

##  RETRIEVE (row = texttype.#row)
##  WHERE length(texttype.#data) = 7

    DISPLAY "Row found = " ROW.
```

VMS

```
DATA DIVISION.
WORKING-STORAGE SECTION

##   01 ROW           PIC S9(4) USAGE COMP.
##   01 DATA          PIC X(7).
##   DECLARE.

PROCEDURE DIVISION.
BEGIN.

*     Set up the table for testing
##    CREATE texttype (#row = i2, #data = text(10))

*     The first APPEND adds the string "abc"
*     (blanks truncated)
##    APPEND TO texttype (#row = 1, #data = data)

*     The second APPEND adds the string "abc ", with
*     4 trailing   blanks
##    APPEND TO texttype (#row = 2, #data = NOTRIM(data))

*     The RETRIEVE will get the second row because
*     the NOTRIM function  in the previous APPEND
*     caused trailing blanks to be inserted as  data.

##    RETRIEVE (row = texttype.#row)
##       WHERE length(texttype.#data) = 7
      DISPLAY "Row found = " ROW.
```

Dynamically Built Param Statements

The **param** feature dynamically builds EQUEL statements. EQUEL/COBOL does not currently support **param** versions of statements. **Param** statements are supported in EQUEL/C and EQUEL/Fortran.

Runtime Error Processing

This section describes a user-defined EQUEL error handler.

Programming for Error Message Output

By default, all Ingres and forms system errors are returned to the EQUEL program, and default error messages are printed on the standard output device. As discussed in the *QUEL Reference Guide* and the *Forms-based Application Development Tools User Guide*, you can also detect the occurrences of errors by means of the program using the **inquire_ingres** and **inquire_frs** statements. (Use the latter for checking errors after forms statements. Use **inquire_ingres** for all other EQUEL statements.)

Because COBOL does not allow the use of local function arguments, you cannot use the EQUEL error handling procedure described in the *QUEL Reference Guide* that entails creating an error-handling function and passing its address to the Ingres runtime routine **IIseterr()**.

Instead, you can simulate the operations of an error function using the **set_ingres** statement. You can also use this statement to suppress the default messages. The general syntax of the **set_ingres** statement is:

set_ingres(*EQUELconst=val*{,*EQUELconst=val*}**)**

where

- *EQUELconst* is one of the valid EQUEL constants to which status information can be assigned.
- *val* is the value that is assigned to the EQUEL status flag. This may be a constant or a program variable containing the value to assign.

The following table presents two of the legal values and types for *EQUELconst.*

Set_ingres Constant Values

Constant	Value	Description
errormode	**integer**	Indicates to EQUEL whether to display Ingres error messages or not, using the normal Ingres error printing routines. Assigning 0 to **errormode** silences Ingres error printing. Assigning 1 to **errormode** normalizes Ingres error printing. The default value is 1, and error messages are printed.
errordisp	**integer**	Displays the error message that corresponds to the last error encountered by Ingres or EQUEL. Assigning 1 to errordisp displays the message.

With these commands, you can perform most of the error-handling functions available in other EQUEL languages. However, the **errordisp** constant used with the **set_ingres** statement displays the last Ingres or EQUEL error on the standard output device. If you wish to obtain the text of the last error while at the same time suppressing the default printing of messages, you can use the **set_ingres** statement in conjunction with the **inquire_ingres** statement, as follows:

```
*   Silence EQUEL errors messages
##  SET_INGRES (ERRORMODE = 0)

##  REPLACE employee (empname = "Fred")

*   Check to see if an error occurred.
*   Assume that "errorvar" is a numeric data item that
*   has been declared to EQUEL.
```

```
##  INQUIRE_INGRES (errorvar = ERRORNO)

    IF (errorvar  0) THEN
       PERFORM ERROR-PROC
    END-IF

. . .

ERROR-PROC.

*   Assume that "errorstr" is an alphanumeric data item
*   that   has been declared to EQUEL.
##  INQUIRE_INGRES (errorstr = ERRORTEXT)

    DISPLAY "Error text is" errorstr

    . . .
```

You should be aware that the **set_ingres** method of error handling makes it difficult for the program to detect conversion errors when returning data to the COBOL program in a **retrieve** loop. This happens because the check for the error condition can only be made at the completion of the **retrieve** loop.

The following example demonstrates how the **set_ingres** command may be used to process error message printing:

```
    DATA DIVISION.
    WORKING-STORAGE DIVISION.
##  01 REAL-VAR PIC S9(8)V9(6) USAGE COMP-3 VALUE 0.469.
##  01 CON-ERR PIC Z999.
##  DECLARE.

PROCEDURE DIVISION.
MAIN.
##  INGRES "Equeldb".

##  /* Create a temporary table */
##  CREATE temp (ccol = c2, icol = i2)
##  APPEND TO temp (ccol = "ab", icol = 1)

##  /*
##  Silence EQUEL error messages and do a replacement
##  that causes a conversion error between a c2
##  column and a numeric data item.
##  */

##  SET_INGRES (ERRORMODE = 0)
##  REPLACE temp (icol = 2) WHERE temp.ccol = REAL-VAR

##  /* Check the EQUEL conversion error and display
##  the message. */
##  INQUIRE_INGRES (CON-ERR = ERRORNO)
    DISPLAY "Conversion error was ", CON-ERR.

##  DESTROY temp
*   Continue program here.
```

A more practical example is a handler to catch deadlock errors. For deadlock, a reasonable handling technique in most applications is to suppress the normal error message and simply restart the transaction.

The following EQUEL program executes a Multi-Query Transaction and handles Ingres errors, including restarting the transaction on deadlock.

In this example, Ingres error messages are silenced, using the **set_ingres** command with the **errormode** option. When errors do occur, they are tested for deadlock. Note that if deadlock does occur, the transaction is restarted automatically without your knowledge.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MQTHANDLER.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.

##  01 INGERR              PIC S9(4) USAGE COMP.
    01 ERR-DISP            PIC ZZZZ99 USAGE DISPLAY.
##  01 ERR-TEXT            PIC X(80).
    01 ING-DEADLOCK        PIC S9(4) USAGE COMP.

##  DECLARE.

    PROCEDURE DIVISION.
    MAIN.
*   Initialize test data & Ingres table, and silence
*   Ingres errors.
    PERFORM INIT-DB

*   Perform transaction, it includes appending
*   and replacing data.
    PERFORM PROCESS-DATA

*   End multi-statement transaction and Ingres
*   interface.
##  END TRANSACTION
##  DESTROY ITEM
##  EXIT
    STOP RUN.

    INIT-DB.
*   Start up Ingres and create a temporary
*   test relation.
##  INGRES testdb
##  CREATE item (name=c10, number=i4)

*   Silence Ingres error messages
##  SET_INGRES (ERRORMODE = 0).

    PROCESS-DATA.
*   Begin a multi-query transaction and reset
*   deadlock flag.
##  BEGIN TRANSACTION.

##  APPEND TO item (name = "Barbara", number=38)
    PERFORM DEADLOCK
    IF (ING-DEADLOCK = 1)
        GO TO PROCESS-DATA
    END-IF

##  REPLACE item (number=39) WHERE item.name="Barbara"
    PERFORM DEADLOCK
    IF (ING-DEADLOCK = 1)
        GO TO PROCESS-DATA
```

```
    END-IF

##  DELETE item WHERE item.number=38
    PERFORM DEADLOCK
    IF (ING-DEADLOCK = 1)
        GO TO PROCESS-DATA
    END-IF.

   DEADLOCK.
*    If the Ingres error is deadlock, the DBMS will
*    automatically abort an existing MQT. If the error is
*    not deadlock, abort  the transaction and the program.

##   INQUIRE_INGRES (INGERR = ERRORNO)
       IF (INGERR > 0) THEN
           IF (INGERR = 4700) THEN
*    Deadlock has occurred
                MOVE 1 TO ING-DEADLOCK
       ELSE
*    DISPLAY Ingres error message & abort the
*    transaction
               MOVE INGERR TO ERR-DISP
               DISPLAY "Aborting on Error" ERR-DISP
##             INQUIRE_INGRES (ERR-TEXT = ERRORTEXT)
               DISPLAY ERR-TEXT
##             ABORT
##             DESTROY item
##             EXIT
               STOP RUN
           END-IF
     ELSE
*    Reset deadlock flag
         MOVE 0 TO ING-DEADLOCK
     END-IF.
```

Precompiling, Compiling, and Linking an EQUEL Program

This section describes the EQUEL preprocessor for COBOL and the steps required to precompile, compile, and link an EQUEL program.

Generating an Executable Program

Once you have written your EQUEL program, it must be preprocessed to convert the EQUEL statements into COBOL code. This section describes the use of the EQUEL preprocessor. Additionally, it describes how to compile the resulting COBOL code.

The EQUEL Preprocessor Command

The following command line invokes the COBOL preprocessor:

eqcbl {*flags*} {*filename*}

where *flags* are

VMS

-a Accepts and generates output in ANSI format. Use this flag if your source code is in ANSI format and you wish to compile the program with the **cobol** command line qualifier **ansi_format**. The code that the preprocessor generates is also in ANSI format. If you omit this flag, the preprocessor accepts input and generates output in VAX COBOL terminal format. For more information, see Source Code Format in this chapter.

-d Adds debugging information to the runtime database error messages EQUEL generates. The source file name, line number, and the erroneous statement are printed with the error message.

-f[*filename*] Writes preprocessor output to the named file. If the **-f** flag is specified without a *filename*, the output is sent to standard output, one screen at a time. If you omit the **-f** flag, output is given the basename of the input file, with the ".cob" extension.

-l Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename***.lis**, where *file*name is the name of the input file.

-lo Like **-l**, but the generated COBOL code also appears in the listing file.

-n. *ext* Specifies the extension used for filenames in **## include** and **## include inline** statements in the source code. If **-n** is omitted, **include** filenames in the source code must be given the extension ".qcb".

-o Directs the processor not to generate output files for include files.

This flag does not affect the translated **include** statement in the main program. The preprocessor generates a default extension for the translated **include** file statements unless you use the **-o**.*ext* flag.

-o. *ext* Specifies the extension the preprocessor gives to both the translated include statements in the main program and the generated output files. If this flag is not provided, the default extension is ".qcb".If you use this flag in combination with the **-o** flag, then the preprocesssor generates the specified extension for the translated **include** statements, but does not generate

	new output files for the **include** statements.
-s	Reads input from standard input and generates COBOL code to standard output. This is useful for testing unfamiliar statements. If the **-l** option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Control D** (UNIX) or **Control Z** (VMS).
-w	Prints warning messages.
-?	Shows what command line options are available for **eqcbl**.

The EQUEL/COBOL preprocessor assumes that input files are named with the extension ".qcb". To override this default, specify the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated COBOL statements with the same name and the extension ".cbl" (UNIX) or ".cob" (VMS).

If you enter only the command, without specifying any flags or a filename, a list of flags available for the command is displayed.

The following table presents the options available with **eqcbl**.

Eqcbl Command Options

Command	Comment
eqcbl file1	Preprocesses "file1.qcb" to: file1.cbl (UNIX) file1.cob (VMS)
eqcbl -l file2.xc	Creates listing "file2.lis" and preprocesses "file2.xcb" to: file2.cbl (UNIX) file2.cob (VMS)
eqcbl -s	Accepts input from standard input and write generated code to standard output
eqcbl -ffile3.out file3	Preprocesses "file3.qcb" to "file3.out"
eqcbl	Displays a list of available flags

Source Code Format

Format Considerations—UNIX

The preprocessor produces MF COBOL II source code in ANSI format using certain conventions. Indicators for comments and continued string literals are placed in column 7. The 01 level number for data declarations known to the preprocessor is output in Area A, starting at column 8. All other generated statements are placed in Area B, starting at column 12. No statements generated extend beyond column 72. COBOL statements and declarations unknown to the preprocessor appear in the preprocessor output file unchanged from the input file.

The preprocessor does not generate any code in columns 1 - 6 (the Sequence Area). Do not, however, precede EQUEL statements with sequence numbers—the ## signal must always appear in the first two columns. Also, although the preprocessor never *generates* code beyond column 72 no matter which format is used, it does *accept* code in columns 73 - 80. Therefore, anything placed in that area on an EQUEL line must be valid EQUEL code.

Format Considerations—VMS

The preprocessor can produce source code written in either VAX COBOL terminal format or ANSI format. The default is terminal format; if you require ANSI format, you should indicate so with the **-a** flag on the preprocessor command line. The COBOL code that the preprocessor generates for EQUEL statements follows the format convention you have chosen.

When you specify the **-a** flag, the preprocessor generates code using certain conventions. Indicators for comments and continued string literals are placed in column 7. The 01 level number for data declarations known to the preprocessor is output in Area A, starting at column 8. All other generated statements are placed in Area B, starting at column 12. No statements generated extend beyond column 72. Note that COBOL statements and declarations unknown to the preprocessor appear in the preprocessor output file unchanged from the input file.

The preprocessor does not generate any code in columns 1 - 6 (the Sequence Area) when you specify the **-a** flag. Do not, however, precede EQUEL statements with sequence numbers—the ## signal must always appear in the first two columns. Also, although the preprocessor never *generates* code beyond column 72 no matter which format you use, it does *accept* code in columns 73 - 80. Therefore, anything placed in that area on an EQUEL line must be valid EQUEL code.

The COBOL Compiler—VMS

The preprocessor generates COBOL code. You should use the VMS **cobol** (VAX-11 C) command to compile this code.

The following example preprocesses and compiles the file "test1." Both the EQUEL preprocessor and the COBOL compiler assume the default extensions.

```
$ eqcbl test1
$ cobol/list test1
```

Note: Check the Readme file for any operating system specific information on compiling and linking EQUEL/COBOL programs.

Linking an EQUEL Program

EQUEL programs require procedures from several VMS shared libraries in order to run properly. After preprocessing and compiling an EQUEL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
$ link dbentry.obj,-
   ii_system:[ingres.files]equel.opt/opt
```

It is recommended that you do not explicitly link in the libraries referenced in the EQUEL.OPT file. The members of these libraries change with different releases of Ingres. Consequently, you can be required to change your link command files in order to link your EQUEL programs.

Assembling and Linking Precompiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *QUEL Reference Guide.* To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. After creating the MACRO file this way, you can assemble it into linkable object code with the VMS command

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry") by listing it in the link command, as in the following example:

```
$   link formentry,-
    empform.obj,-
    ii_system:[ingres.files]equel.opt/opt
```

Linking an EQUEL Program without Shared Libraries

While the use of shared libraries in linking EQUEL programs is recommended for optimal performance and ease-of-maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by EQUEL are listed in the equel.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
$   link dbentry,-
    ii_system:[ingres.files]equel.noshare/opt
```

Incorporating Ingres into the Micro Focus RTS—UNIX

Before you can run any EQUEL/COBOL program, you must create a new Micro Focus Runtime System (RTS), linked with the Ingres libraries. This will enable your EQUEL/COBOL programs to access the necessary Ingres routines at runtime.

If you are not sure your COBOL RTS is linked to the Ingres libraries, you can perform a simple test. Preprocess, compile, and run a simple EQUEL/COBOL program that connects and disconnects from Ingres. For example, the simple test file "test.qcb" could include the lines:

```
## INGRES dbname
## EXIT dbname
```

If your COBOL RTS is not linked to the Ingres libraries, you will receive the COBOL runtime error number 173 when you run the program:

```
$ eqcbl test.qcb
$ cob test.cbl
$ cobrun test
    Load error: file 'IIingopen'
    error code: 173, pc=1A, call=1, seg=0
    173  Called program file not found in drive/directory
```

Note: Check the Readme file for any operating system specific information on compiling and linking ESQL/COBOL programs.

The COBOL Runtime System

To compile the code produced by the preprocessor, use the Micro Focus **cob** command.

The following example preprocesses and compiles the file "test1."

```
$ eqcbl test1.qbc
$ cob test1.cbl
```

When you use the **cob** command to compile the generated COBOL code using the **cob** command, the compiler issues the following informational message:

```
       01 III4-1 PIC S9(9) USAGE COMP-5 SYNC VALUE 0.
**209-I*****************************************
**     COMP-5 is machine specific format.
```

COMP-5 is an Ingres-compatible numeric data type (see COBOL Variables and Data Types in this chapter), and a data item of that type is included in the Ingres system COPY file. You can ignore this warning or you can suppress it using the **cob** compiler directive or command line flag:

```
cob -C warning=1
```

Also, because the program will be run through the COBOL interpreter that is linked to the Ingres runtime system, do not modify the default values of the COBOL compiler **align** and **ibmcomp** directives. To run your EQUEL/COBOL test program, use the **ingrts** command (an alias to your Ingres-linked RTS):

```
ingrts test1
```

For more information on building and linking the Interpreter (or RTS), see Incorporating Ingres into the Micro Focus RTS—UNIX in this chapter.

Building an Ingres RTS Without the Ingres FRS

If you are using the COBOL screen utilities and do not need the Ingres forms runtime system (FRS) incorporated into your COBOL runtime support module, then you can link the RTS exclusively for database activity.

This section describes how to provide the COBOL RTS with all Ingres runtime routines.

Create a directory in which you want to store the Ingres-linked RTS. For example, if the COBOL root directory is "/usr/lib/cobol", you may want to add a new directory "/usr/lib/cobol/ingres" to store the Ingres/COBOL RTS. From that new directory, issue the commands that extract the Ingres Micro Focus support modules, link the Ingres COBOL RTS, and supply an alias to run the new program. The shell script shown below performs all of these steps. Note that "$II_SYSTEM" refers to the path-name of the Ingres root directory on your system:

```
#
# These 2 steps position you in the directory in which
# you want to build the RTS
#
mkdir /usr/lib/cobol/ingres
cd /usr/lib/cobol/ingres
#
# Extract 2 Ingres Micro Focus COBOL support modules
#
ar xv $II_SYSTEM/ingres/lib/libingres.a iimfdata.o
ar xv $II_SYSTEM/ingres/lib/libingres.a iimflibq.o
#
# Now link the new Ingres COBOL RTS (this example c
```

```
# calls it "ingrts")
#
cob -x -e "" -o ingrts                      \
       iimfdata.o iimflibq.o                \
       $II_SYSTEM/ingres/lib/libingres.a    \
       -lc -lm
#
# Provide an alias to run the new program (distribute to
#  RTS users)
#
alias ingrts /usr/lib/cobol/ingrts
```

Ingres shared libraries are available on some Unix platforms. To link with these shared libraries replace "libingres.a" in the **cob** command with:

```
-L $II_SYSTEM/ingres/lib -linterp.1 -lframe.1 -lq.1 \
     -lcompat.1
```

To verify if your release supports shared libraries check for the existence of any of these four shared libraries in the $II_SYSTEM/ingres/lib directory. For example:

```
ls -l $II_SYSTEM/ingres/lib/libq.1.*
```

Since the resulting RTS is quite large, the temporary holding directory required by COBOL may need to be reset. By default, this directory is set to "/usr/tmp". If you are issued "out of disk space" errors during the linking of the Ingres/COBOL RTS, you should consult your COBOL Programmer's Reference Manual to see how to modify the TMPDIR environment variable.

Note that you may need to specify other system libraries in addition to the "-lm" library on the **cob** command. The libraries required are the same as those needed to link an EQUEL/C program. The library names may be added to the last line of the **cob** command shown above. For example, if the "inet" and the "inetd" system libraries are required, the last line of the **cob** command would be:

```
-lc -lm -linet -linetd
```

At this point you are ready to run your EQUEL/COBOL program.

Building an RTS with the Ingres FRS

If you are using the Ingres forms system in your EQUEL/COBOL programs, then you must include the Ingres FRS in the RTS. The link script shown below builds an RTS that includes the Ingres FRS:

```
#
# Optional: Assume you are in an appropriate directory
# as described in the previous section.
#
cd /usr/lib/cobol/ingres
#
# Extract 3 Ingres Micro Focus support modules
#
ar xv $II_SYSTEM/ingres/lib/libingres.a iimfdata.o
ar xv $II_SYSTEM/ingres/lib/libingres.a iimflibq.o
```

```
ar xv $II_SYSTEM/ingres/lib/libingres.a iimffrs.o
#
# Now link the new Ingres COBOL RTS (example calls
# it "ingfrs")
#
cob -x -e "" -o ingfrs \
       iimfdata.o iimflibq.o iimffrs.o \
       $II_SYSTEM/ingres/lib/libingres.a \
       -lc -lm
# Provide an alias to run the new program (distribute
# to RTS users)
#
alias ingfrs /usr/lib/cobol/ingfrs
```

Here, too, you may be required to specify other system libraries on the **cob** command line. For information about how to specify other system libraries on the **cob** command line, see Building an Ingres RTS Without the Ingres FRS in this chapter.

Including External Compiled Forms in the RTS

The description of how to build an Ingres RTS that can access the Ingres forms system does not include a method with which to include compiled forms into the RTS. (Compiled forms are pre-compiled form objects that do not need to be retrieved from the database. Refer to your language reference manual for a description of precompiled forms.) Since the compiled forms are external objects (in object code), you must link them into your RTS.

Because some UNIX platforms allow you to use the Micro Focus **EXTERNAL** clause to reference objects linked into your RTS and some do not, two procedures are given here. The first procedure describes how to include external compiled forms in the RTS on a platform that does permit the use of the **EXTERNAL** clause. The second procedure describes how to perform this task on a platform that does not allow **EXTERNAL** data items to reference objects linked to the RTS.

Procedure 1

Use this procedure if your platform accepts the **EXTERNAL** clause to reference objects linked into your RTS.

1. Build and compile the form(s) in VIFRED.

   When you compile a form in VIFRED, you are prompted for the name of the file, and VIFRED then creates the file in your directory describing the form in C.

2. Compile the C file into object code:

   ```
   $ cc -c formfile.c
   ```

3. Link the compiled forms into your RTS by modifying the **cob** command line to include the object code files for the forms. List the files before listing the system libraries that will be linked.

For example:

```
cob -x -e "" -o ingfrs \
    iimfdata.o iimflibq.o iimffrs.o \
    form1.o form2.o \
    ...
```

Procedure 2

Use this procedure if your platform does not allow you to use the Micro Focus EXTERNAL clause to reference objects linked into your RTS. The extra step forces the external object to be loaded into your RTS and allows access to it through your EQUEL/COBOL program.

1. Build and compile the form(s) in VIFRED.

   When you compile a form in VIFRED, you are prompted for the name of the file, and VIFRED then creates the file in your directory describing the form in C.

2. Compile the C file into object code:

   ```
   $ cc -c formfile.c
   ```

3. Write a small EQUEL/C procedure that just references and initializes the form(s) to the Ingres FRS using the **addform** statement.

   Make sure that the name of the procedure follows conventions allowed for externally called names. For example, external names may be restricted to 14 characters on some versions of COBOL.

   For example:

   ```
   add_form1()
   {
   ## extern int   *form1;
   ## ADDFORM form1
   }
   add_form2()
   {
   ## extern int   *form2;
   ## ADDFORM form2
   }
   ```

4. Build the object code for the initialization of the compiled form(s):

   ```
   $ eqc filename.qc
   $ cc -c filename.c
   ```

   where *filename.qc* is the name of the file containing the procedure written in Step 3.

5. Link the compiled form(s) and the initialization references to the form(s) into your RTS by modifying the **cob** command line to include the object files for the forms and the procedure. Specify the object files before the list of system libraries.

For example:

```
cob -x -e "" -o ingfrs                  \
    iimfdata.o iimflibq.o iimffrs.o     \
    filename.o form1.o form2.o          \
            ...
```

where *filename.o* is the name of object code file resulting from Step 4, containing the initialization references to the forms "form1" and "form2."

6. Replace the **addform** statement in your source program with a COBOL CALL statement to the appropriate C initialization procedure. For example, what would have been:

   ```
   ## ADDFORM form1
   ```

   becomes:

   ```
   CALL "add_form1".
   ```

To illustrate this procedure, assume you have compiled two forms in VIFRED, "empform" and "deptform," and need to be able to access them from your EQUEL/COBOL program without incurring the overhead (or database locks) of the **forminit** statement. After compiling them into C from VIFRED, turn them into object code:

```
$ cc -c empform.c deptform.c
```

Now create an EQUEL/C file, "addforms.qc", that includes a procedure (or two) that initializes each one using the **addform** statement:

```
add_empform()
{
## extern int *empform;
## ADDFORM empform
}

add_deptform()
{
## extern int *deptform;
## ADDFORM deptform
}
```

Now build the object code for the initialization of these 2 compiled forms:

```
$ eqc addforms.qc
$ cc -c addforms.c
```

Then link the compiled forms and the initialization references to those forms into your RTS:

```
cob -x -e "" -o ingfrs \
        iimfdata.o iimflibq.o iimffrs.o \
        addforms.o empform.o deptform.o \
        ...
```

Finally, be sure to replace the appropriate **addform** statements in your source code with COBOL **CALL** statements.

Note, of course, that you may store all your compiled forms in an archive library that will not require the constant modification of a link script. The sample applications (see Sample Applications in this chapter) were built using such a method that included a single file, "addforms.qc", and an archive library, "compforms.a", that included all the compiled forms referenced in the sample applications.

If, at a later time you are able to reference **EXTERNAL** data items directly from your COBOL source code, then the intermediate step of creating an EQUEL/C **addform** procedure can be skipped, and your compiled form is declared as an **EXTERNAL PIC S9(9) COMP-5** data-item in your EQUEL/COBOL source code:

```
##  01   empform IS EXTERNAL PIC S9(9) USAGE COMP-5.
         ...
##       ADDFORM empform
```

The external object code for each form must still be linked into the RTS but there is no need to write an EQUEL/C intermediate file, or call an external C procedure to initialize the compiled form for you.

Include File Processing

The EQUEL **include** statement provides a means to include external files in your program's source code. Its syntax is:

include *filename*

Filename is a double quoted string constant specifying a file name or an environment variable that points to the file name. You must use the default extension ".qcb" on names of **include** files unless you override this requirement by specifying a different extension with the **-n** flag of the **eqcbl** command.

This statement is normally used to include variable declarations although it is not restricted to such use. For more details on the **include** statement, see the *QUEL Reference Guide*.

The included file is preprocessed and an output file with the same name but with the default output extension ".cbl" for UNIX or ".lib" for VMS is generated. You can override this default output extension with the **-o***.ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified include output file. If you use the **-o** flag, with no extension, no output file is generated for the include file. In VMS this is useful for program libraries that are using VMS MMS dependencies.

UNIX

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## INCLUDE "employee.qcb"
```

is preprocessed to the COBOL statement:

```
COPY "employee.cbl"
```

and the file "employee.qcb" is translated into the COBOL file "employee.cbl.".

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
## INCLUDE "mydecls"
```

The name "mydecls" is defined as a system environment variable pointing to the file "/src/headers/myvars.qcb" by means of the following command at the shell level:

```
$ setenv mydecls /src/headers/myvars.qcb
```

Assume now that "inputfile" is preprocessed with the command:

```
$ eqcbl -o.hdr inputfile
```

The command line specifies ".hdr" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the COBOL statement:

```
COPY "/src/headers/myvars.hdr"
```

and the COBOL file "/src/header/myvars.hdr" is generated as output for the original include file, "/src/header/myvars.qbc."

You can also specify include files with a relative path. For example, if you preprocess the file "/src/mysource/myfile.qcb," the EQUEL statement:

```
## INCLUDE "../headers/myvars.qcb"
```

is preprocessed to the COBOL statement:

```
COPY "../header/myvars.cbl"
```

and the COBOL file "/src/headers/myvars.cbl" is generated as output for the original include file, "/src/headers/myvars.qcb."

VMS

If you use both the **-o**.*ext* flag and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program, but does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## INCLUDE "employee.qcb"
```

is preprocessed to the COBOL statement:

```
COPY "employee.lib"
```

and the employee.qcb file is translated into the COBOL file "employee.lib."

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
## INCLUDE "mydecls"
```

The name "mydecls" is defined as a system logical name pointing to the file "dra1:[headers]myvars.qcb" by means of the following command at the DCL level:

```
$ define mydecls dra1:[headers]myvars.qcb
```

Assume now that "inputfile" is preprocessed with the command:

```
$ eqcbl -o.hdr inputfile
```

The command line specifies ".hdr" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the COBOL statement:

```
COPY "dra1:[headers]myvars.hdr"
```

and the COBOL file "dra1:[headers]myvars.hdr" is generated as output for the original include file, "dra1:[headers]myvars.qcb."

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.qcb," the EQUEL statement:

```
## INCLUDE "[-.headers]myvars.qcb"
```

is preprocessed to the COBOL statement:

COPY "[-.headers]myvars.lib"

Including Source Code with Labels

Some EQUEL statements generate labels in the output code. If you include a file containing such statements, you must be careful to include the file only once in a given COBOL scope. Otherwise, you may find that the compiler later issues COBOL warning or error messages to the effect that the generated labels are defined more than once in that scope.

The statements that generate labels are the **retrieve** statement and all the EQUEL/FORMS block-type statements, such as **display** and **unloadtable**.

Coding Requirements for Writing EQUEL Programs

This section describes the coding requirements for writing EQUEL programs.

Comments Embedded in COBOL Output

Each EQUEL statement generates one comment and a few lines of COBOL code. You may find that the preprocessor translates 50 lines of EQUEL into 200 lines of COBOL. This may result in confusion about line numbers when you debug the original source code. To facilitate debugging, a comment corresponding to the original EQUEL source precedes each group of COBOL statements associated with a particular statement. (A comment precedes only *executable* EQUEL statements.) Each comment is one line long and describes the file name, line number, and type of statement in the original source file.

Embedding Statements In IF and PERFORM Blocks

The preprocessor can produce several COBOL statements for a single EQUEL statement. In most circumstances, you can simply nest the statements in the scope of a COBOL **IF** or **PERFORM** statement.

There are some EQUEL statements for which the preprocessor generates COBOL paragraphs and paragraph names. These statements are:

retrieve
display
formdata
unloadtable
submenu

These statements cannot be nested in the scope of a COBOL **IF** or **PERFORM** statement because of the paragraph names the preprocessor generates for them.

Another consequence of these generated paragraphs is that they can terminate the scope of a local COBOL paragraph, thus modifying the intended flow of control. For example, a paragraph generated by the preprocessor in a source paragraph can cause the program to return prematurely to the statement following the **PERFORM** statement that called the source paragraph. To ensure that control does not return prematurely, you must use the **THROUGH** clause in the **PERFORM** statement.

The following example demonstrates the use of **PERFORM-THROUGH** and an **EXIT** paragraph to force correct control flow:

UNIX

```
DATA DIVISION.
WORKING-STORAGE SECTION.

##  01    ENAME         PIC X(20).
##  DECLARE

    PROCEDURE DIVISION.
    BEGIN.

*  Initialization of program

*  Note the THROUGH clause to ensure correct control
*  flow.
     PERFORM UNLOAD-TAB THROUGH END-UNLOAD.

* User code

    UNLOAD-TAB.
* This paragraph includes a paragraph generated by the
* preprocessor

##   UNLOADTABLE Empform Employee (ENAME = Lastname)
##   {
##       APPEND TO person (name = ENAME)
##   }

* This paragraph-name and EXIT statement causes control
* to pass back to the caller's scope
     END-UNLOAD.
        EXIT.
```

VMS

```
* This paragraph-name causes control to pass back to
* the callers scope
END-UNLOAD.
USER-PARAGRAPH.
*Program continues
```

COBOL Periods and EQUEL Statements

You can optionally follow an EQUEL statement with a COBOL separator period although the preprocessor never *requires* that a period follow an EQUEL statement. If the period *is* present at the end of an EQUEL statement, however, the last COBOL statement that the preprocessor generates for that statement also ends with a period. Therefore, you should follow the same guidelines for using the separator period in EQUEL statements as in COBOL statements. For instance, do not add a period at the end of an EQUEL statement occurring in the middle of the scope of a COBOL **IF** or **PERFORM** statement. If you include the separator period in such a case, you prematurely end the scope of the COBOL statement. Similarly, when an EQUEL statement is the *last* statement in the scope of a COBOL **IF**, you *must* follow it with a period (or, alternatively, an **END-IF**) to terminate the scope of the **IF**.

For example:

```
     IF ERR-NO > 0 THEN
* Do not use a separating period in the middle of an IF
* statement.
##         MESSAGE "You cannot update the database"
* Be sure to use a separating period at the end of
* an IF statement.
##         SLEEP 2.
```

In the example above, the absence of the period after the first **message** statement causes the preprocessor to generate code *without* the separator period, thus preserving the scope of the **IF** statement. The period following the **sleep** statement causes the preprocessor to generate code *with* a final separator period, terminating the scope of the **IF**.

An EQUEL Statement that Does Not Generate Code

The **declare cursor** statement does not generate any COBOL code. Do not code this statement as the only statement in any COBOL construct that does not allow *null* statements. For example, coding a **declare cursor** statement as the only statement in a COBOL **IF** statement causes compiler errors:

```
    IF USING-DATABASE=1 THEN
##         DECLARE CURSOR empcsr FOR
## RETRIEVE (employee.ename)
       ELSE
       DISPLAY "You have not accessed the database".
```

The code generated by the preprocessor would be:

```
   IF USING-DATABASE=1 THEN
   ELSE
        DISPLAY "You have not accessed the database".
```

which is an illegal use of the COBOL **ELSE** clause.

Efficient Code Generation

This section describes the COBOL code generated by the EQUEL/COBOL preprocessor.

COBOL Strings and EQUEL Strings

UNIX

COBOL stores string and character data in a machine-dependent data item. The EQUEL runtime routines are written in another language (C) that verifies lengths of strings by the location of a null (LOW-VALUE) byte. Consequently, COBOL strings must be converted to EQUEL runtime strings before the call to the runtime routine is made.

In some languages, EQUEL generates a nested function call that accepts as its argument the character data item and returns the address of the EQUEL null-terminated string. COBOL does not have nested function calls, and simulating this would require two expensive COBOL statements. EQUEL/COBOL knows the context of the statement and, in most cases, will **MOVE** the COBOL string constant or data item in a known area that has already been null-terminated. This extra statement is cheaper than the nested function call of other languages, as it generates a single machine instruction. Even though your COBOL-generated code may look wordier and longer than other EQUEL-generated code, it is actually as efficient. 

VMS

VAX/VMS COBOL stores string and character data in a machine-dependent descriptor. The EQUEL runtime routines are written in another language (C) that verifies lengths of strings by the location of a null (LOW-VALUE) byte. Consequently, COBOL strings must be converted to EQUEL runtime strings before the call to the runtime routine is made.

In some languages, EQUEL generates a nested function call that accepts as its argument the VAX string descriptor and returns the address of the EQUEL null-terminated string. COBOL does not have nested function calls, and simulating this would require two expensive COBOL statements. EQUEL/COBOL knows the context of the statement, and in most cases will **MOVE** the COBOL string constant or data item in a known area that is already null terminated. This extra statement is cheaper than the nested function call of other languages, as it generates a single machine instruction. Even though your COBOL-generated code can look wordier and longer than other EQUEL-generated code, it is actually as efficient. 

COBOL IF-THEN-ELSE Blocks

There are some statements that normally generate an **IF-THEN-ELSE** construct in other languages that instead generate **IF-GOTO** constructs in COBOL. The reason for this is that there is no way to ensure that no EQUEL-generated (or programmer-generated) period will appear in an **IF** block. Consequently, in order to allow any statement in this scope, EQUEL generates an **IF-GOTO** construct. The code generated by EQUEL for this construct is actually very similar to the code generated by any compiler for an **IF-THEN-ELSE** construct and as efficient.

COBOL Function Calls

COBOL supports function calls with the **USING** clause for UNIX or the **GIVING** clause for VMS. This allows a function to return a value into a declared data item. EQUEL generates many of these statements by assigning the return values into internally declared data items, and then checking the result of the function by checking the value of the data item. This is less efficient than other languages that check the return value of a function using its implicit value (stored in a register). The generated COBOL has the overhead of assigning the value to a data item. An EQUEL/COBOL generated function call that tests the result can look like:

UNIX

```
CALL "IIFUNC" USING IIRESULT
IF (IIRESULT = 0) THEN ...
```

VMS

```
CALL "IIFUNC" GIVING IIRESULT
IF (IIRESULT = 0) THEN ...
```

EQUEL/COBOL Preprocessor Errors

To correct most errors, you may wish to run the EQUEL preprocessor with the listing (**-l**) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to the COBOL language, see the next section.

Preprocessor Error Messages

The following is a list of errors messages specific to the COBOL language.

E_E40001 Ambiguous qualification of COBOL data item "%0c"

Explanation: This data item is not sufficiently qualified in order to distinguish it from another data item. It is likely that the data item is an elementary member of a COBOL record or group.

To avoid reference ambiguity qualify the data item further by using IN or OF. When using COBOL table subscripts (with parenthesis), the subscripted item must be unambiguous when the left parenthesis is processed. The preprocessor generates code using the most recently declared instance of the ambiguous data item.

E_E40002 Unsupported COBOL numeric **picture** string "%0c".

Explanation: An invalid picture character was encountered while processing a numeric picture string. A numeric picture string may include the following:

S
9
(
)
V

The preprocessor treats the data item as though it was declared:

```
PICTURE S9(8) USAGE COMP.
```

E_E40003 COMP picture "%0c" requires too many storage bytes. Try USAGE COMP-3.

Explanation: The COMPUTATIONAL data type must fit into a maximum of 4 bytes. Numeric integers of more than 9 digits require VAX quad-word integer storage (8 bytes), which is incompatible with the Ingres internal runtime data types.

Try reducing the picture string or declaring the data item as COMP-3 or COMP-2 which is compatible with Ingres floating-point data. An exception is made to allow non-scaled 10-digit numeric picture strings (PICTURE S(10) USAGE COMP), which is representable by a 4-byte integer.

E_E40004 No **## declare** before first EQUEL statement "%0c".

Explanation: You must issue the **## declare** statement before the first embedded statement. The preprocessor generates code that references data items declared in a file copied by the **## declare** statement. Without issuing the **## declare** statement, the COBOL compiler will not accept those references.

E_E40005 "%0c" is not an elementary data item. Records cannot be used.

Explanation: In this usage, COBOL records or tables cannot be used. In order to use this data item you must refer to an elementary data item that is a member of the record, or an element of the COBOL table.

E_E40006 COBOL declaration level %0c is out of bounds.

Explanation: Only levels 01 through 49 and 77 are accepted for COBOL data item declarations. Level numbers outside of this range will be treated as though they are level 01.

Syntax errors caused in leading clauses of a COBOL declaration may cascade and generate this error message for the **occurs** and **value** clauses of the erroneous declaration.

E_E40007 Data item requires a picture string in this **usage**.

Explanation: The specified **usage** clause requires a COBOL **picture** string in order to determine preprocessor data item type information. Not all usage clauses require a **picture** string. Data items with **usage comp**, comp-3 and **display** do require a **picture** string. If no **picture** string is specified the preprocessor will treat the data item as though it was declared:

```
PICTURE X(10) USAGE DISPLAY.
```

E_E40008 Data item on level %0c has no parent of lesser level.

Explanation: A data item declared on a level that is greater than the level of the most recently declared data item is considered to be a subordinate member of that group. The previous level, therefore, must be the level number of a COBOL record or group declaration. This is typical with a COBOL record containing a few elementary data items.

A data item declared on a level that is less than the level of the most recently declared data item is considered to be on the same level as the "parent" of that data item. Level numbers violating this rule will be treated as though they are level 01.

E_E40009 Keyword **picture** and the describing string must be on the same line.

Explanation: When the preprocessor scans the COBOL **picture** string, it must find the **picture** keyword and the corresponding string description on the same line in the source file. The **picture** word and the string can be separated by the **is** keyword. The preprocessor will treat the declaration as though there was no **picture** clause.

E_E4000A "%0c" is not a legally declared data item.

Explanation: The specified data item was not declared but has been used in place of a COBOL variable in an embedded statement.

E_E4000B Unsupported **picture** "%0c" is numeric-display. **usage comp** assumed.

Explanation: Some versions of the COBOL preprocessor do not support numeric display data items. For example:

```
PICTURE S9(8) USAGE DISPLAY.
```

If this is the case, you should use COMPUTATIONAL data items and assign to and from **display** items before using the data item in embedded statements.

E_E4000C COBOL **occurs** clause is not allowed on level 01.

Explanation: The **occurs** clause must be used with a data item that is declared on a level greater than 01. This error is only a warning, and treats the data item correctly (as though declared as a COBOL table). A warning may also be generated by the COBOL compiler.

E_E4000D EQUEL/COBOL does not support **param** target lists.

Explanation: This feature is not documented and should not be used with EQUEL COBOL.

E_E4000E **Picture** "%0c" is too long. The maximum length is %1c.

Explanation: COBOL **picture** strings must not exceed the maximum length specified in the error message. Try to collapse consecutive occurrences of the same **picture** symbol into a "repeat count." For example:

```
PICTURE S99999999 becomes PICTURE S9(8)
```

E_E4000F **Picture** "%0c" contains non-integer repeat count, %1c.

Explanation: A COBOL "repeat count'' in a **picture** string was either too long or was not an integer. The preprocessor treats the data item as though declared with a **picture** with a repeat count of 1. For example:

```
S9(1) or X(1).
```

E_E40011 **Usage** type "%0c" is not supported.

Explanation: This **usage** type is currently not supported.

E_E40012 **Picture** "%0c" has two sign symbols (S).

Explanation: The specified numeric **picture** string has two sign symbols. The preprocessor will treat the data item as though it was declared:

```
PICTURE S9(8) USAGE COMP.
```

E_E40013 Picture "%0c" has two decimal point symbols (V).

Explanation: The specified numeric **PICTURE** string has two decimal point symbols. The preprocessor will treat the data item as though it was declared:

```
PICTURE S9(8) USAGE COMP.
```

E_E40014 Missing quotation mark on continued string literal.

Explanation: The first non-blank character of a continued string literal must be a quotation mark in the indicator area. A missing quotation mark in the continued string literal or the wrong quotation mark will generate this error.

E_E40015 COBOL data item "%0c" is a table and must be subscripted.

Explanation: The data item is a COBOL table and must be subscripted in order to yield an elementary data item to retrieve or set Ingres data.

E_E40016 COBOL data item "%0c" is not a table and must not be subscripted.

Explanation: You have included subscripts when referring to a data item that was not declared as a COBOL table.

E_E40017 Duplicate COBOL data declaration "%0c" clause found.

Explanation: You have included either a duplicate USAGE, PICTURE, or OCCURS data declaration clause when declaring a data item.

Sample Applications

This section contains sample applications.

UNIX and VMS—The Department-Employee Master/Detail Application

This section contains a sample master/detail application that uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Department information is stored in program variables. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo (the Toberesolved database table, which is described later) to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second is for the Employee table. The **create** statements used to create the tables are shown below. The cursors retrieve all the information in their respective tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, both from the Department table and the Employee table, is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates.

Each section of code is commented for the purpose of the application and to clarify some of the uses of the EQUEL statements. The program illustrates table creation, multi-query transactions, all cursor statements and direct updates. For purposes of brevity, error handling on data manipulation statements is simply to close down the application.

For readability, all EQUEL reserved words are in uppercase.

The two **create** statements describing the Employee and Department database tables are shown at the start of the program.

UNIX

```
##    CREATE dept
##     (name        = c12,      /* Department name */
##      totsales    = money,    /* Total sales */
##      employees   = i2)       /* Number of employees */

##    CREATE employee
##     (name        = c20,      /* Employee name */
##      age         = i1,       /* Employee age */
##      idno        = i4,       /* Unique employee id */
##      hired       = date,     /* Date of hire */
##      dept        = c10,      /* Employee department */
##      salary      = money)    /* Yearly salary */

        IDENTIFICATION DIVISION.
        PROGRAM-ID. EXPENSE-PROCESS.

        ENVIRONMENT DIVISION.

        DATA DIVISION.

        WORKING-STORAGE SECTION.
##      DECLARE

        *Cursor loop control
##      01 NO-ROWS                  PIC S9(2) USAGE COMP.

        * Minimum sales of department
##      01 MIN-DEPT-SALES           PIC S9(5)V9(2) USAGE COMP
##                                  VALUE IS 50000.00.

        * Minimum employee salary
##      01 MIN-EMP-SALARY           PIC S9(5)v9(2) USAGE COMP
##                                  VALUE IS 14000.00.

        * Age above which no salary-reduction will be made
```

```
##      01 NEARLY-RETIRED           PIC S9(2) USAGE COMP
##                                  VALUE IS 58.

        * Salary-reduction percentage
##      01 SALARY-REDUC             PIC S9(1)V9(2) USAGE COMP
##                                  VALUE IS 0.95.

        * Record corresponding to the "dept" table.
##      01 DEPT.
##         02 DNAME                 PIC X(12).
##         02 TOTSALES              PIC S9(7)V9(2) USAGE COMP.
##         02 EMPLOYEES             PIC S9(4) USAGE COMP.

        * Record corresponding to the "employee" table
##      01 EMP.
##         02 ENAME                 PIC X(20).
##         02 AGE                   PIC S9(2) USAGE COMP.
##         02 IDNO                  PIC S9(8) USAGE COMP.
##         02 HIRED                 PIC X(26).
##         02 SALARY                PIC S9(6)V9(2) USAGE COMP.
##         02 HIRED-SINCE-85        PIC S9(4) USAGE COMP.

        * Count of employees terminated.
##      01 EMPS-TERM                PIC S99 USAGE COMP.

        * Indicates whether the employee's dept was deleted
##      01 DELETED-DEPT             PIC S9 USAGE COMP.

        * Error message buffer used by CHECK-ERRORS.
##      01 ERRBUF                   PIC X(200).

        * Error number
##      01 ERRNUM                   PIC S9(8) USAGE COMP.

        * Formatting values for output
##      01 DEPT-OUT.
##          02 FILLER               PIC X(12) VALUE "Department: ".
##          02 DNAME-OUT            PIC X(12).
##          02 FILLER               PIC X(13) VALUE "Total Sales: ".
##          02 TOTSALES-OUT         PIC $,$$$,$$9.9(2) USAGE DISPLAY.
##          02 DEPT-FORMAT          PIC X(19).

##      01 EMP-OUT.
##          02 FILLER               PIC XX VALUE SPACES.
##          02 TITLE                PIC X(11).
##          02 IDNO-OUT             PIC Z9(6) USAGE DISPLAY.
##          02 FILLER               PIC X VALUE SPACE.
##          02 ENAME-OUT            PIC X(20).
##          02 AGE-OUT              PIC Z9(2) USAGE DISPLAY.
##          02 FILLER               PIC XX VALUE SPACES.
##          02 SALARY-OUT           PIC $$$,$$9.9(2) USAGE DISPLAY.
##          02 FILLER               PIC XX VALUE SPACES.
##          02 DESCRIPTION          PIC X(24).

**
* Procedure Division
*
*       Initialize the database, process each department and
*       terminate the session.
**

        PROCEDURE DIVISION.
        EXAMPLE SECTION.
        XBEGIN.

        DISPLAY "Entering application to process expenses".
```

```
        PERFORM INIT-DB THRU END-INITDB.
        PERFORM PROCESS-DEPTS THRU END-PROCDEPTS.
        PERFORM END-DB THRU END-ENDDB.
        DISPLAY "Successful completion of application".
        STOP RUN.

**
* Paragraph: INIT-DB
*
*       Start up the database, and abort if there is an error
*       Before processing employees, create the table for
*       employees who losetheir department,
*       "toberesolved". Initiate the multi-statement
*       transaction.
**

        INIT-DB.

##      INGRES "personnel"

      * Silence Ingres error printing
##      SET_EQUEL (ERRORMODE = 0)

        DISPLAY "Creating ""To_Be_Resolved"" table".

##      CREATE toberesolved
##            (#name    = char(20),
##             #age     = smallint,
##             #idno    = integer,
##             #hired   = date,
##             #dept    = char(10),
##             #salary  = money)

##      INQUIRE_EQUEL (ERRNUM = ERRORNO)
        IF ERRNUM NOT = 0 THEN
##          INQUIRE_INGRES (ERRBUF = ERRORTEXT)
            DISPLAY "Fatal error on creation:"
            DISPLAY ERRBUF
##          EXIT
            STOP RUN
        END-IF.

##      BEGIN TRANSACTION

        END-INITDB.
            EXIT.

**
* Paragraph: END-DB
*
*       Closes off the multi-statement transaction and access to
*       the database after successful completion of the application
**

        END-DB.

##      END TRANSACTION
##      EXIT

        END-ENDDB.
                    EXIT.

**
* Paragraph: PROCESS-DEPTS
*
*       Scan through all the departments, processing each one.
```

```
*       If the department has made less than $50,000 in sales, then
*       the department is dissolved. For each department process
*       all the employees (they may even be moved to another
        table).
*       If an employee was terminated, then update the department's
*       employee counter.
**

        PROCESS-DEPTS.

##      RANGE OF d IS #dept

##      DECLARE CURSOR deptcsr FOR
##            RETRIEVE (d.#name, d.#totsales, d.#employees)
##            FOR DIRECT UPDATE OF (#name, #employees)

##      OPEN CURSOR deptcsr
        PERFORM CHECK-ERRORS.

        MOVE 0 TO NO-ROWS.
        PERFORM UNTIL NO-ROWS = 1

##            RETRIEVE CURSOR deptcsr (DNAME, TOTSALES, EMPLOYEES)

##            INQUIRE_EQUEL (NO-ROWS = ENDQUERY)

              IF NO-ROWS = 0 THEN

*                   Did the department reach minimum sales?

                    IF TOTSALES < MIN-DEPT-SALES THEN
##                        DELETE CURSOR deptcsr
                          PERFORM CHECK-ERRORS
                          MOVE 1 TO DELETED-DEPT
                          MOVE " -- DISSOLVED --" TO DEPT-FORMAT
                    ELSE
                          MOVE 0 TO DELETED-DEPT
                          MOVE SPACES TO DEPT-FORMAT
                    END-IF

*                   Log what we have just done

                    MOVE DNAME TO DNAME-OUT
                    MOVE TOTSALES TO TOTSALES-OUT
                    DISPLAY DEPT-OUT

*                   Now process each employee in the department

                    PERFORM PROCESS-EMPLOYEES THRU
                                        END-PROCEMPLOYEES
                    MOVE 0 TO NO-ROWS

*                   If some employees were terminated, record this
*                   fact


                    IF EMPS-TERM > 0 AND DELETED-DEPT = 0 THEN
##                        REPLACE CURSOR deptcsr
##                             (#employees = EMPLOYEES - EMPS-TERM)
                          PERFORM CHECK-ERRORS
                    END-IF

          END-IF

        END-PERFORM.
```

```
##      CLOSE CURSOR deptcsr

        END-PROCDEPTS.
           EXIT.

**
* Paragraph: PROCESS-EMPLOYEES
*
*       Scan through all the employees for a particular department.
*       Based on given conditions the employee may be terminated,
*       or given a salary reduction:
*       1.If an employee was hired since 1985 then the employee is
*         terminated.
*       2.If the employee's yearly salary is more than the minimum
*         company wage of $14,000 and the employee is not close to
*         retirement (over 58 years of age), then the employee take
*         takes a 5% salary reduction.
*       3.If the employee's department is dissolved and the
*         employee is not terminated, then the employee is moved
*         into the "toberesolved" table.
**

        PROCESS-EMPLOYEES.

*       Note the use of the Ingres functions to find out
*       who was hired since 1985.

##      RANGE OF e IS #employee

##      DECLARE CURSOR empcsr FOR
##              RETRIEVE (e.#name, e.#age, e.#idno, e.#hired ##
                  e.#salary,res = int4(
##                 interval("days",e.#hired - date("01-jan-1985"))
##                     )
##                )
##                WHERE e.#dept = DNAME
##                FOR DIRECT UPDATE OF (#name, #salary)

##      OPEN CURSOR empcsr
        PERFORM CHECK-ERRORS.

*       Record how many employees are terminated

        MOVE 0 TO EMPS-TERM.

        MOVE 0 TO NO-ROWS.
        PERFORM UNTIL NO-ROWS = 1

##            RETRIEVE CURSOR empcsr
##                (ENAME, AGE, IDNO, HIRED, SALARY, HIRED-SINCE-85)

##            INQUIRE_EQUEL (NO-ROWS = ENDQUERY)

              IF NO-ROWS = 0 THEN

                IF HIRED-SINCE-85 > 0 THEN

##                 DELETE CURSOR empcsr
                   PERFORM CHECK-ERRORS
                   MOVE "Terminated:" TO TITLE
                   MOVE "Reason: Hired since 1985." TO DESCRIPTION
                   ADD 1 TO EMPS-TERM

              ELSE

*               Reduce salary if not nearly retired
```

```
                IF SALARY > MIN-EMP-SALARY THEN
                  IF AGE < NEARLY-RETIRED THEN
##                         REPLACE CURSOR empcsr
##                                (#salary = #salary * SALARY-REDUC)
                          PERFORM CHECK-ERRORS
                          MOVE "Reduction: " TO TITLE
                          MOVE "Reason: Salary." TO DESCRIPTION
                  ELSE
*                        Do not reduce salary
                         MOVE "No Changes:" TO TITLE
                         MOVE "Reason: Retiring." TO DESCRIPTION
                  END-IF

*                                Leave employee alone
                                 ELSE
                    MOVE "No Changes:" TO TITLE
                    MOVE "Reason: Salary." TO DESCRIPTION
                  END-IF

*                 Was employee's department dissolved?
                  IF DELETED-DEPT = 1 THEN
##                  RANGE OF e IS #employee
##                  APPEND TO toberesolved (e.all)
##                         WHERE e.#idno = IDNO
                    PERFORM CHECK-ERRORS
##                  DELETE CURSOR empcsr
                  END-IF
                END-IF

*                  Log the employee's information
                  MOVE IDNO TO IDNO-OUT
                  MOVE ENAME TO ENAME-OUT
                  MOVE AGE TO AGE-OUT
                  MOVE SALARY TO SALARY-OUT
                  DISPLAY EMP-OUT

                  END-IF

              END-PERFORM.

##            CLOSE CURSOR empcsr

              MOVE 0 TO ERRNUM.

              END-PROCEMPLOYEES.
                EXIT.

**
* Paragraph: CHECK-ERRORS
*
*      This paragraph serves as an error handler called any time
*      after INIT-DB has successfully completed
*      In all cases, it prints the cause of the error, and
*      aborts the transaction, backing out changes.
*      Note that disconnecting from the database will
*      implicitly close any open cursors too. If an error is found
*      the application is aborted.
**

      CHECK-ERRORS.

      MOVE 0 TO ERRNUM.
##    INQUIRE_EQUEL (ERRNUM = ERRORNO)
      IF ERRNUM NOT = 0 THEN
*         Restore Ingres error printing
```

```
##        SET_EQUEL (ERRORMODE = 1)
##        INQUIRE_INGRES (ERRBUF = ERRORTEXT)
##        ABORT
##        EXIT
          DISPLAY "Closing Down because of database error:"
          DISPLAY ERRBUF
          STOP RUN
      END-IF.
```

VMS

```
           ##     CREATE dept
           ##        (name      = c12,     /* Department name */
           ##         totsales  = money,   /* Total sales */
           ##         employees = i2)     /* Number of employees */

          ##     CREATE employee
          ##        (name      = c20,     /* Employee name */
          ##         age       = i1,      /* Employee age */
          ##         idno      = i4,      /* Unique employee id */
          ##         hired     = date,    /* Date of hire */
          ##         dept      = c10,     /* Employee department */
          ##         salary    = money)   /* Yearly salary */

IDENTIFICATION DIVISION.
PROGRAM-ID. EXPENSE-PROCESS.

ENVIRONMENT DIVISION.

DATA DIVISION.

WORKING-STORAGE SECTION.
##      DECLARE

* Cursor loop control
##      01 NO-ROWS            PIC S9(2) USAGE COMP.

* Minimum sales of department
##      01 MIN-DEPT-SALES     USAGE COMP-2 VALUE IS 50000.00.

* Minimum employee salary
##      01 MIN-EMP-SALARY     USAGE COMP-2 VALUE IS 14000.00.

* Age above which no salary-reduction will be made
##      01 NEARLY-RETIRED     PIC S9(2) USAGE COMP VALUE IS 58.

* Salary-reduction percentage
##      01 SALARY-REDUC       USAGE COMP-1 VALUE IS 0.95.

* Indicates whether "toberesolved" table exists in INIT-DB
* paragraph.
##      01 FOUND-TABLE        PIC S9 USAGE COMP.

* Record corresponding to the "dept" table.
##      01            DEPT.
##         02 NAME            PIC X(12).
##         02 TOTSALES        USAGE COMP-2.
##         02 EMPLOYEES       PIC S9(4) USAGE COMP.
```

```
* Record corresponding to the "employee" table
##      01            EMP.
##         02 NAME            PIC X(20).
##         02 AGE             PIC S9(2) USAGE COMP.
##         02 IDNO            PIC S9(8) USAGE COMP.
##         02 HIRED           PIC X(26).
##         02 SALARY          USAGE COMP-2.
##         02 HIRED-SINCE-85  PIC S9(4) USAGE COMP.

* Count of employees terminated.
##      01 EMPS-TERM          PIC S99 USAGE COMP.

* Indicates whether the employee's dept was deleted
##      01 DELETED-DEPT       PIC S9 USAGE COMP.

* Error message buffer used by CLOSE-DOWN
##      01 ERRBUF             PIC X(100).

* Error number
##      01 ERRNUM             PIC S9(8) USAGE COMP.

* Formatting values for output
##      01            DEPT-OUT.
##         02 FILLER          PIC X(12) VALUE "Department: ".
##         02 DNAME           PIC X(12).
##         02 FILLER          PIC X(13) VALUE "Total Sales: ".
##         02 TOTSALES-OUT    PIC $,$$$,$$9.9(2) USAGE DISPLAY.
##         02 DEPT-FORMAT     PIC X(19).

##     01  EMP-OUT.
##         02 FILLER          PIC XX VALUE SPACES.
##         02 TITLE           PIC X(11).
##         02 IDNO-OUT        PIC Z9(6) USAGE DISPLAY.
##         02 FILLER          PIC X VALUE SPACE.
##         02 ENAME           PIC X(20).
##         02 AGE-OUT         PIC Z9(2) USAGE DISPLAY.
##         02 FILLER          PIC XX VALUE SPACES.
##         02 SALARY-OUT      PIC $$$,$$9.9(2) USAGE DISPLAY.
##         02 FILLER          PIC XX VALUE SPACES.
##         02 DESCRIPTION     PIC X(24).

**
* Procedure Division
*
*     Initialize the database, process each department and
*     terminate the session.
**

PROCEDURE DIVISION.
SBEGIN.

         DISPLAY "Entering application to process expenses".
         PERFORM INIT-DB THRU END-INITDB.
         PERFORM PROCESS-DEPTS THRU END-PROCDEPTS.
         PERFORM END-DB THRU END-ENDDB.
         DISPLAY "Successful completion of application".
         STOP RUN.

**
* Paragraph: INIT-DB
*
*     Start up the database, and abort if there is an error.
*     Before processing employees, create the table for employees
*     who lose their department, "toberesolved". Initiate the
*     multi-statement transaction.
**
```

```
INIT-DB.

##    INGRES "personnel"

* Silence INGRES error printing

##    SET_EQUEL (ERRORMODE = 0)

      DISPLAY "Creating ""To_Be_Resolved"" table".

##    CREATE toberesolved
##        (#name   = char(20),
##         #age    = smallint,
##         #idno   = integer,
##         #hired  = date,
##         #dept   = char(10),
##         #salary = money)

##    INQUIRE_EQUEL (ERRNUM = ERRORNO)
      IF ERRNUM NOT = 0 THEN
##              INQUIRE_INGRES (ERRBUF = ERRORTEXT)
                DISPLAY "Fatal error on creation:"
                DISPLAY ERRBUF
##              EXIT
                STOP RUN
      END-IF.

##    BEGIN TRANSACTION

END-INITDB.

**
* Paragraph: END-DB
*
*     Closes off the multi-statement transaction and access to
*     the database after successful completion of the application.
**

END-DB.

##    END TRANSACTION
##    EXIT

END-ENDDB.
**
* Paragraph: PROCESS-DEPTS
*
*     Scan through all the departments, processing each one.
*     If the department has made less than $50,000 in sales, then
*     the department is dissolved. For each department process
*     all the employees (they may even be moved to another table).
*     If an employee was terminated, then update the department's
*     employee counter.
**

PROCESS-DEPTS.

##    RANGE OF d IS #dept

##    DECLARE CURSOR deptcsr FOR
##              RETRIEVE (d.#name, d.#totsales, d.#employees)
##              FOR DIRECT UPDATE OF (#name, #employees)

##    OPEN CURSOR deptcsr
      PERFORM CHECK-ERRORS.
```

```
      MOVE 0 TO NO-ROWS.
      PERFORM UNTIL NO-ROWS = 1

##               RETRIEVE CURSOR deptcsr
##                        (NAME IN DEPT, TOTSALES, EMPLOYEES)

##               INQUIRE_EQUEL (NO-ROWS = ENDQUERY)
                 IF NO-ROWS = 0 THEN

* Did the department reach minimum sales?

                     IF TOTSALES < MIN-DEPT-SALES THEN

##                          DELETE CURSOR deptcsr
                            PERFORM CHECK-ERRORS

                           MOVE 1 TO DELETED-DEPT
                            MOVE " -- DISSOLVED --" TO DEPT-FORMAT

                     ELSE

                              MOVE 0 TO DELETED-DEPT
                              MOVE "" TO DEPT-FORMAT

                     END-IF

* Log what we have just done

                     MOVE NAME IN DEPT TO DNAME
                     MOVE TOTSALES TO TOTSALES-OUT
                     DISPLAY DEPT-OUT

* Now process each employee in the department

                   PERFORM PROCESS-EMPLOYEES THRU END-PROCEMPLOYEES
                     MOVE 0 TO NO-ROWS

* If some employees were terminated, record this fact

                     IF EMPS-TERM > 0 AND DELETED-DEPT = 0 THEN
##                        REPLACE CURSOR deptcsr
##                             (#employees = EMPLOYEES - EMPS-TERM)
                        PERFORM CHECK-ERRORS
                     END-IF

                END-IF

          END-PERFORM.

##        CLOSE CURSOR deptcsr

END-PROCDEPTS.

**
* Paragraph: PROCESS-EMPLOYEES
*
*        Scan through all the employees for a particular department.
*        Based on given conditions the employee may be terminated,
*        or given a salary reduction:
*        1.  If an employee was hired since 1985 then the employee
*            is terminated.
*        2.  If the employee's yearly salary is more than the
*            minimum company wage of $14,000 and the employee
*            is not close to retirement (over 58 years of age),
*            then the employee takes a 5% salary reduction
*        3.  If the employee's department is dissolved and the
```

```
*            employee is not terminated, then the employee
*            is moved into the "toberesolved" table.
**

PROCESS-EMPLOYEES.

* Note the use of the INGRES functions to find out who was hired
* since 1985.

##        RANGE OF e IS #employee

##        DECLARE CURSOR empcsr FOR
##              RETRIEVE (e.#name, e.#age, e.#idno, e.#hired,
##                  e.#salary, res = int4(
##                  interval("days",e.#hired - date("01-jan-1985"))
##                    )
##              )
##              WHERE e.#dept = NAME IN DEPT
##              FOR DIRECT UPDATE OF (#name, #salary)

##        OPEN CURSOR empcsr
          PERFORM CHECK-ERRORS.

* Record how many employees terminated

          MOVE 0 TO EMPS-TERM.

          MOVE 0 TO NO-ROWS.
          PERFORM UNTIL NO-ROWS = 1

##         RETRIEVE CURSOR empcsr
##          (NAME IN EMP, AGE, IDNO, HIRED, SALARY, HIRED-SINCE-85)

##              INQUIRE_EQUEL (NO-ROWS = ENDQUERY)

                IF NO-ROWS = 0 THEN

                    IF HIRED-SINCE-85 > 0 THEN

##                      DELETE CURSOR empcsr
                        PERFORM CHECK-ERRORS

                        MOVE "Terminated:" TO TITLE
                     MOVE "Reason: Hired since 1985."TO DESCRIPTION
                        ADD 1 TO EMPS-TERM

                    ELSE

* Reduce salary if not nearly retired

                      IF SALARY > MIN-EMP-SALARY THEN

                           IF AGE < NEARLY-RETIRED THEN

##                            REPLACE CURSOR empcsr
##                            (#salary = #salary * SALARY-REDUC)
                              PERFORM CHECK-ERRORS
                              MOVE "Reduction: " TO TITLE
                              MOVE "Reason: Salary."TO DESCRIPTION
                       ELSE

* Do not reduce salary
                             MOVE "No Changes:" TO TITLE
                             MOVE "Reason: Retiring."TO DESCRIPTION
                          END-IF
```

```
* Leave employee alone

                    ELSE
                            MOVE "No Changes:" TO TITLE
                            MOVE "Reason: Salary."TO DESCRIPTION
                    END-IF

* Was employee's department dissolved?

                    IF DELETED-DEPT = 1 THEN

##                          RANGE OF e IS #employee
##                          APPEND TO toberesolved (e.all)
##                                  WHERE e.#idno = IDNO
                            PERFORM CHECK-ERRORS
##                          DELETE CURSOR empcsr

                    END-IF

                  END-IF

* Log the employee's information

                  MOVE IDNO TO IDNO-OUT
                  MOVE NAME IN EMP TO ENAME
                  MOVE AGE TO AGE-OUT
                  MOVE SALARY TO SALARY-OUT
                  DISPLAY EMP-OUT

               END-IF

          END-PERFORM.

##        CLOSE CURSOR empcsr

          MOVE 0 TO ERRNUM.

END-PROCEMPLOYEES.

**
* Paragraph: CHECK-ERRORS
*
*        This paragraph serves as an error handler called any time
*        after INIT-DB has successfully completed. In all cases,
*        it prints the cause of the error, and aborts the
*        transaction, backing out changes. Note that disconnecting
*         from the database will implicitly close any open cursors
*         too is aborted. If an error is found the application
**

CHECK-ERRORS.

      MOVE 0 TO ERRNUM.
##    INQUIRE_EQUEL (ERRNUM = ERRORNO)
      IF ERRNUM NOT = 0 THEN
*             Restore INGRES error printing
##            SET_EQUEL (ERRORMODE = 1)
##            INQUIRE_INGRES (ERRBUF = ERRORTEXT)
##            ABORT
##            EXIT
              DISPLAY "Closing Down because of database error:"
              DISPLAY ERRBUF
              STOP RUN
      END-IF.
```

UNIX and VMS—The Employee Query Interactive Forms Application

This EQUEL/FORMS application uses a form in **query** mode to view a subset of the Employee table in the Personnel database. An Ingres query qualification is built at runtime using values entered in fields of the form "empform."

The objects used in this application are:

Object	Description
personnel	The program's database environment.
employee	A table in the database, with six columns: name (**c20**) age (**i1**) idno (**i4**) hired (**date**) dept (**c10**) salary (**money**)
empform	A VIFRED form with fields corresponding in name and type to the columns in the Employee database table. The name and idno fields are used to build the query and are the only updatable fields. Empform is a compiled form.

The application is driven by a **display** statement that allows the runtime user to enter values in the two fields that will build the query. The **Build_Query** and **Exec_Query** procedures make up the core of the query that is run as a result. Note the way the values of the query operators determine the logic used to build the **where** clause in **Build_Query**. The **retrieve** statement encloses a **submenu** block that allows the user to step through the results of the query.

No updates are performed on the values retrieved, but any particular employee screen may be saved in a log file through the **printscreen** statement.

For readability, all EQUEL reserved words are in uppercase.

UNIX

The following **create** statement describes the format of the Employee database table:

```
##  CREATE employee
##    (name    = c20,     /* Employee name */
##     age     = i1,      /* Employee age */
##     idno    = i4,      /* Unique employee id */
##     hired   = date,    /* Date of hire */
##     dept    = c10,     /* Employee department */
##     salary  = money)   /* Annual salary */

        IDENTIFICATION DIVISION.
```

```
        PROGRAM-ID. EMPLOYEE-QUERY.

        ENVIRONMENT DIVISION.

        DATA DIVISION.
        WORKING-STORAGE SECTION.
##      DECLARE

*       For WHERE clause qualification
##      01      WHERE-CLAUSE         PIC X(100).

*       Query operators
##      01       NAME-OP             PIC S9(8) USAGE COMP.
##      01       ID-OP               PIC S9(8) USAGE COMP.

*       Were rows found?
##      01 ROWS PIC S9(8) USAGE COMP.

##      01  FORM-VALUES.
##          02 ENAME                 PIC X(20).
##          02 EIDNO                 PIC S9(8) USAGE COMP.
##          02 EAGE                  PIC S9(2) USAGE COMP.
##          02 EHIRED                PIC X(25).
##          02 ESALARY               PIC S9(6)V9(2) USAGE COMP-3.
##          02 DISP-IDNO             PIC ZZZZZ9.

*       Note: Compiled forms are not yet accepted as EXTERNAL due
*       to restrictions noted in the chapter
*       that describes how to link the RTS with compiled forms.
*       Consequently the declarations of external form
*       objects and the corresponding ADDFORM statement
*       have been commented out and replaced by a CALL
*       "add_formname" statement.
##      01 empform PIC S9(9) USAGE COMP-5 IS EXTERNAL.

*       Query operator table that maps integer values to string
*       query operators

        01 OPER-MASKS.
            02 FILLER VALUE "= "  PIC X(3).
            02 FILLER VALUE "!= " PIC X(3).
            02 FILLER VALUE "< "  PIC X(3).
            02 FILLER VALUE "> "  PIC X(3).
            02 FILLER VALUE "<= " PIC X(3).
            02 FILLER VALUE ">= " PIC X(3).
         01 OPER-TABLE REDEFINES OPER-MASKS.
            02 OPER OCCURS 6 TIMES  PIC X(3).

         PROCEDURE DIVISION.
         EXAMPLE SECTION.
         XBEGIN.

*        Initialize WHERE clause qualification buffer to be an
*        Ingres default qualification that is always true

         MOVE "1=1" TO WHERE-CLAUSE.

##       FORMS
##       MESSAGE "Accessing Employee Query Application..."
##       INGRES "personnel"

* ##     ADDFORM empform
         CALL "add_empform".

##       DISPLAY #empform QUERY
##       INITIALIZE
```

```
##        ACTIVATE MENUITEM "Reset"
##        {
##               CLEAR FIELD ALL
##        }
##        ACTIVATE MENUITEM "Query"
##        {

*               Verify validity of data
##              VALIDATE

                PERFORM BUILD-QUERY THROUGH ENDBUILD-QUERY.
                PERFORM EXEC-QUERY THROUGH ENDEXEC-QUERY.
##        }
##        ACTIVATE MENUITEM "LastQuery"
##        {
               PERFORM EXEC-QUERY THROUGH ENDEXEC-QUERY.
##        }
##             ACTIVATE MENUITEM "End", FRSKEY3
##        {
##               BREAKDISPLAY
##        }
##        FINALIZE

##        ENDFORMS
##        EXIT
          STOP RUN.

**
* Paragraph: BUILD-QUERY
*
*         Build a query from the values in the "name and "idno"
*         fields in "empform."
**

          BUILD-QUERY.

##        GETFORM #empform (
##              ENAME = name, NAME-OP = GETOPER(name),
##              EIDNO = idno, ID-OP = GETOPER(idno)
##        )

*         Fill in the WHERE clause
          MOVE SPACES TO WHERE-CLAUSE.

          IF NAME-OP = 0 AND ID-OP = 0 THEN

                 MOVE "1 = 1" TO WHERE-CLAUSE

          ELSE IF NAME-OP NOT = 0 AND ID-OP NOT = 0 THEN

*                         Query on both fields
                 MOVE EIDNO TO DISP-IDNO
                 STRING "e.name " DELIMITED BY SIZE,
                         OPER(NAME-OP) DELIMITED BY " ",
                         """ DELIMITED BY SIZE,
                         ENAME DELIMITED BY SIZE,
                         """ DELIMITED BY SIZE,
                         " and e.idno " DELIMITED BY SIZE,
                         OPER(ID-OP) DELIMITED BY " ",
                       DISP-IDNO DELIMITED BY SIZE INTO WHERE-CLAUSE
          ELSE IF NAME-OP NOT = 0 THEN

*                Query on the 'name' field
                 STRING "e.name " DELIMITED BY SIZE,
                         OPER(NAME-OP) DELIMITED BY " ",
                         """ DELIMITED BY SIZE,
```

```
                        ENAME DELIMITED BY SIZE,
                        """" DELIMITED BY SIZE INTO WHERE-CLAUSE

        ELSE

*               Query on the 'idno' field
                MOVE EIDNO TO DISP-IDNO
                STRING "e.idno " DELIMITED BY SIZE,
                        OPER(ID-OP) DELIMITED BY " ",
                      DISP-IDNO DELIMITED BY SIZE INTO WHERE-CLAUSE

        END-IF.

        ENDBUILD-QUERY.
          EXIT.
**
* Paragraph: EXEC-QUERY
*
*       Given a query buffer defining a WHERE clause, issue a
*       RETRIEVE to allow the runtime user to browse the employee
*       found with the given qualification.
**

        EXEC-QUERY.

##      RANGE OF e IS employee

##      RETRIEVE (EIDNO = e.idno, ENAME = e.name, EAGE = e.age,
##                EHIRED = e.hired, ESALARY = e.salary)
##                WHERE WHERE-CLAUSE
##      {

*             Put values onto form and display them

##            PUTFORM #empform (
##                   idno = EIDNO, name = ENAME, age = EAGE,
##                   hired = EHIRED, salary = ESALARY)

##            REDISPLAY

##            SUBMENU
##            ACTIVATE MENUITEM "Next", FRSKEY4
##            {

*                Do nothing, and continue with the RETRIEVE loop.
*                The last one will drop out.

##             }
##             ACTIVATE MENUITEM "Save", FRSKEY8
##             {

*                Save screen data in log file
##               PRINTSCREEN (FILE = "query.log")
*                Drop through to next employee

##             }
##             ACTIVATE MENUITEM "End", FRSKEY3
##             {

*                       Terminate the RETRIEVE loop
##                      ENDRETRIEVE

##             }

##      }
```

```
##      INQUIRE_EQUEL (ROWS = ROWCOUNT)
        IF ROWS = 0 THEN
##          MESSAGE "No rows found for this query."
        ELSE
##          CLEAR FIELD ALL
##          MESSAGE "No more rows. Reset for next query."
        END-IF.
##      SLEEP 2

        ENDEXEC-QUERY.
          EXIT.
```

VMS

The **create** statement describing the format of the Employee database table is shown first:

```
##      CREATE employee
##         (name     = c20,      /* Employee name */
##          age      = i1,       /* Employee age */
##          idno     = i4,       /* Unique employee id */
##          hired    = date,     /* Date of hire */
##          dept     = c10,      /* Employee department */
##          salary   = money)    /* Annual salary */

IDENTIFICATION DIVISION.
PROGRAM-ID. EMPLOYEE-QUERY.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
##      DECLARE
* Compiled form
##      01  EMPFORM-ID PIC S9(9) USAGE COMP VALUE IS EXTERNAL empform.

* For WHERE clause qualification
##      01  WHERE-CLAUSE PIC X(100).

* Query operators
##      01  NAME_OP      PIC S9(8) USAGE COMP.
##      01  ID_OP        PIC S9(8) USAGE COMP.

* Were rows found?
##      01  ROWS         PIC S9(8) USAGE COMP.

##      01  FORM_VALUES.
##          02      ENAME PIC X(20).
##          02      EIDNO PIC S9(8) USAGE COMP.
##          02      EAGE PIC S9(2) USAGE COMP.
##          02      EHIRED PIC X(25).
##          02      EDEPT PIC X(10).
##          02      ESALARY USAGE COMP-2.
##          02      DISP_IDNO PIC ZZZZZ9.

* Query operator table that maps integer values to string
* query operators
        01  OPER_MASKS.
            02     FILLER VALUE "= "     PIC X(3).
            02     FILLER VALUE "!= "    PIC X(3).
            02     FILLER VALUE "< "     PIC X(3).
            02     FILLER VALUE "> "     PIC X(3).
            02     FILLER VALUE "<= "    PIC X(3).
            02     FILLER VALUE ">= "    PIC X(3).
        01  OPER_TABLE REDEFINES OPER_MASKS.
            02     OPER OCCURS 6 TIMES   PIC X(3).
```

```
PROCEDURE DIVISION.
SBEGIN.

* Initialize WHERE clause qualification buffer to be a default
* qualification that is always true

      MOVE "1=1" TO WHERE-CLAUSE.

##    FORMS
##    MESSAGE "Accessing Employee Query Application..."
##    INGRES "personnel"

##    ADDFORM EMPFORM-ID

##    DISPLAY #empform QUERY
##    INITIALIZE
##    ACTIVATE MENUITEM "Reset"
##    {
##          CLEAR FIELD ALL
##    }
##    ACTIVATE MENUITEM "Query"
##    {

* Verify validity of data
##             VALIDATE

               PERFORM BUILD-QUERY THROUGH ENDBUILD-QUERY.
               PERFORM EXEC-QUERY THROUGH ENDEXEC-QUERY.
##    }
##    ACTIVATE MENUITEM "LastQuery"
##    {
               PERFORM EXEC-QUERY THROUGH ENDEXEC-QUERY.
##    }
##    ACTIVATE MENUITEM "End", FRSKEY3
##    {
##             BREAKDISPLAY
##    }
##    FINALIZE

##    ENDFORMS
##    EXIT
      STOP RUN.

**
* Paragraph: BUILD-QUERY
*
*     Build a query from the values in the "name and "idno"
*     fields in "empform."
**
BUILD-QUERY.

##    GETFORM #empform (
##             ENAME = name, NAME_OP = GETOPER(name),
##             EIDNO = idno, ID_OP = GETOPER(idno)
##    )

* Fill in the where clause

      IF NAME_OP = 0 AND ID_OP = 0 THEN

             MOVE "1 = 1" TO WHERE-CLAUSE

      ELSE IF NAME_OP NOT = 0 AND ID_OP NOT = 0 THEN

* Query on both fields
```

```
                MOVE EIDNO TO DISP_IDNO

                STRING "e.name " DELIMITED BY SIZE,
                        OPER(NAME_OP) DELIMITED BY " ",
                        """ DELIMITED BY SIZE,
                        ENAME DELIMITED BY SIZE,
                        """ DELIMITED BY SIZE,
                        " and e.idno " DELIMITED BY SIZE,
                        OPER(ID_OP) DELIMITED BY " ",
                        DISP_IDNO DELIMITED BY SIZE INTO WHERE-CLAUSE

     ELSE IF NAME_OP NOT = 0 THEN

* Query on the 'name' field

                STRING "e.name " DELIMITED BY SIZE,
                        OPER(NAME_OP) DELIMITED BY " ",
                        """ DELIMITED BY SIZE,
                        ENAME DELIMITED BY SIZE,
                        """ DELIMITED BY SIZE INTO WHERE-CLAUSE

      ELSE

* Query on the 'idno' field

                MOVE EIDNO TO DISP_IDNO

                STRING "e.idno " DELIMITED BY SIZE,
                        OPER(ID_OP) DELIMITED BY " ",
                        DISP_IDNO DELIMITED BY SIZE INTO WHERE-CLAUSE

      END-IF.

ENDBUILD-QUERY.

**
* Paragraph: EXEC-QUERY
*
*     Given a query buffer defining a WHERE clause, issue a
*     RETRIEVE to allow the runtime user to browse the employee
*     found with the given qualification.
**

EXEC-QUERY.

##    RANGE OF e IS employee

##    RETRIEVE (EIDNO = e.idno, ENAME = e.name, EAGE = e.age,
##          EHIRED = e.hired, ESALARY = e.salary)
##          WHERE WHERE-CLAUSE
##    {

* Put values on to form and display them

##          PUTFORM #empform (
##                    idno = EIDNO, name = ENAME, age = EAGE,
##                    hired = EHIRED, salary = ESALARY)

##          REDISPLAY

##          SUBMENU
##          ACTIVATE MENUITEM "Next", FRSKEY4
##          {
```

```
* Do nothing, and continue with the RETRIEVE loop. The last
* one will drop out.

##          }
##          ACTIVATE MENUITEM "Save", FRSKEY8
##          {

* Save screen data in log file

##                   PRINTSCREEN (FILE = "query.log")

* Drop through to next employee

##          }
##          ACTIVATE MENUITEM "End", FRSKEY3
##          {

* Terminate the RETRIEVE loop

##                   ENDRETRIEVE

##          }

##     }

##     INQUIRE_EQUEL (ROWS = ROWCOUNT)

       IF ROWS = 0 THEN

##              MESSAGE "No rows found for this query."

       ELSE

##              CLEAR FIELD ALL
##              MESSAGE "No more rows. Reset for next query."

       END-IF.
##     SLEEP 2

ENDEXEC-QUERY.
```

UNIX and VMS—The Table Editor Table Field Application

This EQUEL/FORMS application uses a table field to edit the Person table in the Personnel database. It allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate their use and their interaction with an Ingres database.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**c20**) age (**i2**) number (**i4**). Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**c20**) age (**i4**) When initialized, the table field includes the hidden column number (**i4**).

At the start of the application, a **retrieve** statement is issued to load the table field with data from the Person table. Once the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated or deleted. When finished, the values are unloaded from the table field, and, in a multi-statement transaction, the user's updates are transferred back into the Person table.

For readability, all EQUEL reserved words are in uppercase.

UNIX

The following **create** statement describes the format of the Person database table:

```
##   CREATE person
##     (name   = c20,     /* Person name */
##      age    = i2,      /* Age */
##      number = i4       /* Unique id number */

        IDENTIFICATION DIVISION.
        PROGRAM-ID. TABLE-EDITOR.

        ENVIRONMENT DIVISION.

        DATA DIVISION.
        WORKING-STORAGE SECTION.
```

```
##      DECLARE

*       Record corresponds to "person" table
##      01    PERSON-VALUES.
##            02 PNAME           PIC X(20).
##            02 P-AGE           PIC S9(2) USAGE COMP.
##            02 PNUMBER         PIC S9(8) USAGE COMP.

##      01    MAXID              PIC S9(9) USAGE COMP.

*       Table field row states
##      01    STATE              PIC S9 USAGE COMP.
*             Empty or undefined row
              88 ST-UNDEF VALUE 0.
*             Appended by user
              88 ST-NEW VALUE 1.
*             Loaded by program - not updated
              88 ST-UNCHANGED VALUE 2.
*             Loaded by program - since changed
              88 ST-CHANGED VALUE 3.
*             Deleted by program
              88 ST-DELETED VALUE 4.

*       Table field entry information
##      01    T-RECORD           PIC S9(4) USAGE COMP.
##      01    LASTROW            PIC S9 USAGE COMP.

*       Utility buffers
##      01    MSGBUF             PIC X(200).
##      01    RESPBUF            PIC X(20).

*       Status variables
*       Number of rows updated
##      01    UPDATE-ROWS        PIC S9(4) USAGE COMP.
*       Update error from database
##      01    UPDATE-ERROR       PIC S9(2) USAGE COMP.
*       Transaction aborted
##      01    XACT-ABORTED       PIC S9 USAGE COMP.
*       Save changes to database?
#       01    SAVE-CHANGES       PIC S9 USAGE COMP.

        PROCEDURE DIVISION.
        EXAMPLE SECTION.
        XBEGIN.

*       Start up Ingres and the FORMS system

##      INGRES "personnel"

##      FORMS

*       Verify that the user can edit the "person" table

##      PROMPT NOECHO ("Password for table editor: ", RESPBUF)

        IF RESPBUF NOT = "MASTER_OF_ALL" THEN

##            MESSAGE "No permission for task. Exiting..."
##            ENDFORMS
##            EXIT
              STOP RUN

        END-IF.

##      MESSAGE "Initializing Person Form..."
```

```
##      RANGE OF p IS person

##      FORMINIT personfrm

*       Initialize "persontbl" table field with a data set in FILL
*       mode so that the runtime user can append rows. To keep
*       track of events occurring to original rows that will
*       be loaded into the table field, hide
*       the unique person number.

##      INITTABLE personfrm persontbl FILL (number = integer)

        PERFORM LOAD-TABLE THROUGH ENDLOAD-TABLE.

##      DISPLAY personfrm UPDATE
##      INITIALIZE

##      ACTIVATE MENUITEM "Top", FRSKEY5
##      {

*       Provide menu, as well as the system FRS key to scroll
*       to both extremes of the table field

##      SCROLL personfrm persontbl TO 1

##      }

##      ACTIVATE MENUITEM "Bottom", FRSKEY6
##      {
##            SCROLL personfrm persontbl TO END
##      }
##      ACTIVATE MENUITEM "Remove"
##      {

*             Remove the person in the row the user's cursor is on.
*             Record this in the database later.

##            DELETEROW personfrm persontbl
##      }

##      ACTIVATE MENUITEM "Find", FRSKEY7
##      {

*          Scroll user to the requested table field entry. Prompt
*          the user for a name, and if one is typed in loop through
*          the data set searching for it.

##         PROMPT ("Person's name : ", RESPBUF)

           IF RESPBUF = SPACES THEN
##             RESUME FIELD persontbl
           END-IF.

##         UNLOADTABLE personfrm persontbl
##             (PNAME = name, T-RECORD = _RECORD, STATE = _STATE)
##         {

*              Do not compare with deleted rows

               IF PNAME = RESPBUF AND NOT ST-DELETED THEN

##               SCROLL personfrm persontbl TO T-RECORD
##               RESUME FIELD persontbl

               END-IF.
##         }
```

```
*          Fell out of loop without finding name
           STRING "Person """ DELIMITED BY SIZE,
              RESPBUF DELIMITED BY SIZE,
              """ not found in table [HIT RETURN]"
              DELIMITED BY SIZE
              INTO MSGBUF.
##         PROMPT NOECHO (MSGBUF, RESPBUF)

##      }

##      ACTIVATE MENUITEM "Save", FRSKEY8
##      {
##         VALIDATE FIELD persontbl
           MOVE 1 TO SAVE-CHANGES.
##         BREAKDISPLAY
##      }

##      ACTIVATE MENUITEM "Quit", FRSKEY2
##      {
            MOVE 0 TO SAVE-CHANGES.
##          BREAKDISPLAY
##      }
##      FINALIZE

##      MESSAGE "Exiting Person Application..."

        IF SAVE-CHANGES = 0 THEN
##          ENDFORMS
##          EXIT
            STOP RUN
        END-IF.

*       Exit person table editor and unload the table field. If any
*       updates, deletions or additions were made, duplicate these
*       changes in the source table. If the user added new people
*       we must assign a unique person id before
*       returning it to the table. To do this, increment the
*       previously saved maximum id
*       number with each insert.

*       Do all the updates in a transaction (for simplicity,
*       this transaction does not restart on DEADLOCK error: 4700)

##      BEGIN TRANSACTION

        MOVE 0 TO UPDATE-ERROR.
        MOVE 0 TO XACT-ABORTED.

##      UNLOADTABLE personfrm persontbl
##          (PNAME = name, P-AGE = age, PNUMBER = number,
##              STATE = _STATE)
##      {

        IF ST-NEW THEN

*           Appended by user. Insert with new unique id
            ADD 1 TO MAXID
##          REPEAT APPEND TO person (name = @PNAME,
##             age = @P-AGE,
##             number = @MAXID)

        ELSE IF ST-CHANGED THEN

*           Updated by user. Reflect in table
##          REPEAT REPLACE person (name = @PNAME, age = @P-AGE)
```

```
##             WHERE person.number = @PNUMBER

        ELSE IF ST-DELETED THEN

*           Deleted by user, so delete from table. Note that
*           only original rows are saved by the program, and
*           not rows appended at runtime.
##          REPEAT DELETE FROM p WHERE p.number = @PNUMBER

        END-IF.

*       ELSE ST-UNDEFINED or ST-UNCHANGED - No updates

*       Handle error conditions -
*       If an error occurred, then abort the transaction.
*       If no rows were updated then inform user, and prompt for
*       continuation.

##      INQUIRE_INGRES (UPDATE-ERROR = ERRORNO,
##      UPDATE-ROWS = ROWCOUNT)
        IF UPDATE-ERROR NOT = 0 THEN
*           Error
##          INQUIRE_EQUEL (MSGBUF = ERRORTEXT)
##          ABORT
            MOVE 1 TO XACT-ABORTED
##          ENDLOOP
        ELSE IF UPDATE-ROWS = 0 THEN
            STRING  "Person """, PNAME,
                    """ not updated. Abort all updates? "
                    DELIMITED BY SIZE
                    INTO MSGBUF
##          PROMPT  (MSGBUF, RESPBUF)
            IF RESPBUF = "Y" OR RESPBUF = "y" THEN
##                  ABORT
                    MOVE 1 TO XACT-ABORTED
##                  ENDLOOP
            END-IF
        END-IF.

## }

    IF XACT-ABORTED = 0 THEN
*       Commit the updates
##      END TRANSACTION
    END-IF.

*   Terminate the FORMS and Ingres
##  ENDFORMS
#       EXIT

    IF UPDATE-ERROR NOT = 0 THEN
        DISPLAY "Your updates were aborted because of error:"
        DISPLAY MSGBUF
    END-IF.

    STOP RUN.

**
* Paragraph: LOAD-TABLE
*
*   Load the table field from the "person" table. The columns
*   name" and "age" will be displayed, and "number" will be
*   hidden.
**

    LOAD-TABLE.
```

```
##  MESSAGE "Loading Person Information . . ."

*   Fetch the maximum person id number for later use.
*   NOTE: max() will do a sequential scan of the table.

##  RETRIEVE (MAXID = MAX(p.number))

*   Fetch data, and load table field

##  RETRIEVE (PNAME = p.name, P-AGE = p.age, PNUMBER = p.number)
##  {
##       LOADTABLE personfrm persontbl
##           (name = PNAME, age = P-AGE, number = PNUMBER)
##  }

    ENDLOAD-TABLE.
        EXIT.
```

VMS

The **create** statement describing the format of the Person database table appears first:

```
              ##    CREATE person
              ##          (name    = c20,   /* Person name */
              ##           age     = i2,    /* Age */
              ##           number  = i4)    /* Unique id number */
IDENTIFICATION DIVISION.
PROGRAM-ID. TABLE-EDITOR.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
##        DECLARE

* Record corresponds to "person" table

##     01   PERSON-VALUES.
##          02   PNAME            PIC X(20).
##          02   P-AGE            PIC S9(2) USAGE COMP.
##          02   PNUMBER          PIC S9(8) USAGE COMP.

##     01   MAXID                PIC S9(9) USAGE COMP.

* Table field row states

* Empty or undefined row
##     01   ST-UNDEF             PIC S9 USAGE COMP VALUE 0.
* Appended by user
##     01   ST-NEW               PIC S9 USAGE COMP VALUE 1.
* Loaded by program - not updated
##     01   ST-UNCHANGED         PIC S9 USAGE COMP VALUE 2.
* Loaded by program - since changed
##     01   ST-CHANGED           PIC S9 USAGE COMP VALUE 3.
* Deleted by program
##     01   ST-DELETED           PIC S9 USAGE COMP VALUE 4.
* Table field entry information
##     01    STATE               PIC S9 USAGE COMP.
##     01    T-RECORD            PIC S9(4) USAGE COMP.
##     01    LASTROW             PIC S9 USAGE COMP.

* Utility buffers
```

```
##     01    MSGBUF              PIC X(200).
##     01    RESPBUF             PIC X(20).

* Status variables

* Number of rows updated
##    01     UPDATE-ROWS         PIC S9(4) USAGE COMP.
* Update error from database
##    01     UPDATE-ERROR        PIC S9(2) USAGE COMP.
* Transaction aborted
##    01     XACT-ABORTED        PIC S9 USAGE COMP.
* Save changes to database?
##    01     SAVE-CHANGES        PIC S9 USAGE COMP.

PROCEDURE DIVISION.
SBEGIN.
* Start up Ingres and the FORMS system

##        INGRES "personnel"

##        FORMS

* Verify that the user can edit the "person" table

##        PROMPT NOECHO ("Password for table editor: ", RESPBUF)

          IF RESPBUF NOT = "MASTER_OF_ALL" THEN

##           MESSAGE "No permission for task. Exiting..."
##           ENDFORMS
##           EXIT
             STOP RUN

          END-IF.

##        MESSAGE "Initializing Person Form..."

##        RANGE OF p IS person

##        FORMINIT personfrm

* Initialize "persontbl" table field with a data set in FILL mode
* so that the runtime user can append rows. To keep track of
* events occurring to original rows that will be loaded
* into the table field, hide the unique person number.

##        INITTABLE personfrm persontbl FILL (number = integer)

          PERFORM LOAD-TABLE THROUGH ENDLOAD-TABLE.

##        DISPLAY personfrm UPDATE
##        INITIALIZE

##        ACTIVATE MENUITEM "Top", FRSKEY5
##        {

* Provide menu, as well as the system FRS key to scroll
* to both extremes of the table field

##              SCROLL personfrm persontbl TO 1

##        }
```

```
##        ACTIVATE MENUITEM "Bottom", FRSKEY6
##        {
##               SCROLL personfrm persontbl TO END
##        }

##        ACTIVATE MENUITEM "Remove"
##        {

* Remove the person in the row the user's cursor is on.
* Record this in the database later.

##               DELETEROW personfrm persontbl
##        }

##        ACTIVATE MENUITEM "Find", FRSKEY7
##        {

* Scroll user to the requested table field entry. Prompt the
* user for a name, and if one is typed in loop through the
* data set searching for it.

##               PROMPT ("Person's name : ", RESPBUF)

                 IF RESPBUF = "" THEN
##                    RESUME FIELD persontbl
                 END-IF.

##               UNLOADTABLE personfrm persontbl
##               (PNAME = name, T-RECORD = _RECORD, STATE = _STATE)
##               {

* Do not compare with deleted rows

                 IF PNAME = RESPBUF AND STATE NOT = ST-DELETED THEN

##                           SCROLL personfrm persontbl TO T-RECORD
##                             RESUME FIELD persontbl

                    END-IF.
##                  }

* Fell out of loop without finding name

                    STRING "Person """ DELIMITED BY SIZE,
                           RESPBUF DELIMITED BY SIZE,
                           """ not found in table
                           [HIT RETURN]" DELIMITED BY SIZE
                           INTO MSGBUF.

##                  PROMPT NOECHO (MSGBUF, RESPBUF)

##        }

##        ACTIVATE MENUITEM "Save", FRSKEY8
##        {
##                  VALIDATE FIELD persontbl
                    MOVE 1 TO SAVE-CHANGES.
##                  BREAKDISPLAY
##        }

##        ACTIVATE MENUITEM "Quit", FRSKEY2
##        {
                 MOVE 0 TO SAVE-CHANGES.
##               BREAKDISPLAY
##        }
##        FINALIZE
```

```
##        MESSAGE "Exiting Person Application..."

       IF SAVE-CHANGES = 0 THEN
##                 ENDFORMS
##                 EXIT
                   STOP RUN
       END-IF.

*    Exit person table editor and unload the table field.
*    If any updates, deletions or additions were made, duplicate
*    these changes in the source table. If the user added new
*    people we must assign a unique person id before returning
*    it to the table. To do this, increment the previously
*    saved maximum id number with each insert.

*    Do all the updates in a transaction (for simplicity,
*    this transaction does not restart on DEADLOCK error: 4700)

##     BEGIN TRANSACTION

       MOVE 0 TO UPDATE-ERROR.
       MOVE 0 TO XACT-ABORTED.

##     UNLOADTABLE personfrm persontbl
##          (PNAME = name, P-AGE = age, PNUMBER = number,
##           STATE = _STATE)
##     {

            IF STATE = ST-NEW THEN

*    Appended by user. Insert with new unique id

                     ADD 1 TO MAXID

##                     REPEAT APPEND TO person (name = @PNAME,
##                             age = @P-AGE,
##                             number = @MAXID)

             ELSE IF STATE = ST-CHANGED THEN

*    Updated by user. Reflect in table

##             REPEAT REPLACE person (name = @PNAME, age = @P-AGE)
##                      WHERE person.number = @PNUMBER

             ELSE IF STATE = ST-DELETED THEN

*    Deleted by user, so delete from table. Note that only
*    orignal rows are saved by the program, and not rows
*    appended at runtime.

##               REPEAT DELETE FROM p WHERE p.number = @PNUMBER
             END-IF
*    Else UNDEFINED or UNCHANGED - No updates

*    Handle error conditions -
*    If an error occurred, then abort the transaction.
*    If no rows were updated then inform user, and prompt
*    for continuation.

##          INQUIRE_INGRES (UPDATE-ERROR = ERRORNO, UPDATE-ROWS =
##               ROWCOUNT)
```

```
              IF UPDATE-ERROR NOT = 0 THEN
*       Error
##                INQUIRE_EQUEL (MSGBUF = ERRORTEXT)
##                ABORT
                  MOVE 1 TO XACT-ABORTED
##                ENDLOOP

        ELSE IF UPDATE-ROWS = 0 THEN

                  STRING "Person """ PNAME
                          """ not updated. Abort all updates? "
                          DELIMITED BY SIZE
                          INTO MSGBUF
##                PROMPT (MSGBUF, RESPBUF)
                  IF RESPBUF = "Y" OR RESPBUF = "Y" THEN
##                        ABORT
                          MOVE 1 TO XACT-ABORTED
##                        ENDLOOP
                  END-IF

        END-IF

##    }

       IF XACT-ABORTED = 0 THEN

*     Commit the updates

##                  END TRANSACTION

       END-IF.

*     Terminate the FORMS and Ingres

##     ENDFORMS

##     EXIT

       IF UPDATE-ERROR NOT = 0 THEN

              DISPLAY "Your updates were aborted because of error:"
              DISPLAY MSGBUF

       END-IF.

       STOP RUN.
**
* Paragraph: LOAD-TABLE
*
*       Load the table field from the "person" table. The columns
*       "name" and "age" will be displayed, and "number" will be
*       hidden.
**

LOAD-TABLE.

##       MESSAGE "Loading Person Information . . ."

* Fetch the maximum person id number for later use.
* PERFORMANCE NOTE: max() will do a sequential scan of the table.

##       RETRIEVE (MAXID = MAX(p.number))

* Fetch data, and load table field
```

```
##      RETRIEVE (PNAME = p.name, P-AGE = p.age, PNUMBER = p.number)
##      {
##                LOADTABLE personfrm persontbl
##                     (name = PNAME, age = P-AGE, number = PNUMBER)
##      }

ENDLOAD-TABLE.
```

UNIX and VMS—The Professor-Student Mixed Form Application

This EQUEL/FORMS application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The application uses the following objects:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**c25**) pdept (**c10**). See its **create** statement below for a full description.
student	A database table with seven columns: sname (**c25**) sage (**i1**) sbdate (**c25**) sgpa (**f4**) sidno (**i1**) scomment (**text(200**)) sadvisor (**c25**). See the create statement below for a full description. The sadvisor columnm is the join field with the pname column in the Professor table.
masterfrm	The main form has the pname and pdept fields that correspond to the information in the Professor table, and the studenttbl table field. The pdept field is display-only. Masterfrm is a compiled form.
studenttbl	A table field in masterfrm with two columns, sname and sage. When initialized, it also has five more hidden columns corresponding to information in the Student table.

Object	Description
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the sgpa, scomment and sadvisor fields are updatable. All other fields are display-only. Studentfrm is a compiled form.
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the studentfrm form and the studenttbl table field.

The program uses the masterfrm as the general-level master entry, in which data can only be retrieved and browsed, and the studentfrm as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the pname (professor name) field and then selects the **Students** menu operation. The operation fills the displayed and hidden columns of the studenttbl table field with detailed information of the students reporting to the named professor. The user may then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student may be requested by selecting the **Zoom** menu operation. This operation displays the studentfrm form. The fields of studentfrm are filled with values stored in the hidden columns of studenttbl. The user may make changes to three fields (sgpa, scomment, and sadvisor). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

For readability, all EQUEL reserved words are in uppercase.

UNIX

The following two **create** statements describe the Professor and Student database tables:

```
##   CREATE student   /* Graduate student table */
##    (sname     = c25,       /* Name */
##     sage      = i1,        /* Age */
##     sbdate    = c25,       /* Birth date */
##     sgpa      = f4,        /* Grade point average */
##     sidno     = i4,        /* Unique student number */
##     scomment  = text(200), /* General comments */
##     sadvisor  = c25)       /* Advisor's name */

##   CREATE professor    /* Professor table */
##    (pname     = c25,       /* Professor's name */
##     pdept     = c10)       /* Department */

        IDENTIFICATION DIVISION.
        PROGRAM-ID. STUDENT-ADMINISTRATOR.

        ENVIRONMENT DIVISION.

        DATA DIVISION.
        WORKING-STORAGE SECTION.
##      DECLARE
```

```
*       Global grad student record maps to database table
##      01    GRAD.
##             02 SNAME           PIC X(25).
##             02 SAGE            PIC S9(4) USAGE COMP.
##             02 SBDATE          PIC X(25).
##             02 SGPA            PIC S9(3)V9(2) USAGE COMP.
##             02 SIDNO           PIC S9(9) USAGE COMP.
##             02 SCOMMENT        PIC X(200).
##             02 SADVISOR        PIC X(25).

*       Professor info maps to database table
##      01     PROF.
##              02 PNAME          PIC X(25).
##              02 PDEPT          PIC X(10).

*       Row number of last row in student table field
##      01     LASTROW            PIC S9(9) USAGE COMP.

*       Is user on a table field?
##      01     ISTABLE            PIC S9 USAGE COMP.

*       Were changes made to data in student form?
##      01     CHANGED-DATA       PIC S9 USAGE COMP.

*       Did user enter a valid advisor name?
##      01     VALID-ADVISOR      PIC S9 USAGE COMP.

*       Studentfrm loaded?
##      01     LOADFORM           PIC S9 USAGE COMP VALUE IS 0.

*       Local utility buffers
##      01     MSGBUF             PIC X(100).
##      01     RESPBUF            PIC X.
##      01     OLD-ADVISOR        PIC X(25).

*       Note: Compiled forms are not yet accepted as EXTERNAL due
*       to restrictions noted in the chapter that
*       describes how to link the RTS with compiled forms.
*       Consequently the declarations of external form
*       objects and the corresponding ADDFORM statement
*       have been commented out and replaced by a CALL
*       "add_formname" statement.
* ##    01    masterfrm      PIC S9(9) USAGE COMP-5 IS EXTERNAL.
* ##    01    studentfrm     PIC S9(9) USAGE COMP-5 IS EXTERNAL.

**
*       Procedure Division: STUDENT-ADMINISTRATOR
*
*       Start up program, Ingres, and the FORMS system and
*       call Master driver.
**

        PROCEDURE DIVISION.
        EXAMPLE SECTION.
        XBEGIN.

##      FORMS

##      MESSAGE "Initializing Student Administrator . . ."

##      INGRES "personnel"
##      RANGE OF p IS professor, s IS student

        PERFORM MASTER THROUGH END-MASTER.

##      CLEAR SCREEN
```

```
##      ENDFORMS
##      EXIT
        STOP RUN.

**
*       Paragraph: MASTER
*
*           Drive the application, by running "masterfrm", and
*           allowing the user to "zoom" into a selected student.
**

        MASTER.

* ##    ADDFORM masterfrm
        CALL "add_masterfrm".

*       Initialize "studenttbl" with a data set in READ mode.
*       Declare hidden  columns for all the extra fields that
*       the program will display when   more information is
*       requested about a student.
*       Columns "sname" and  "sage" are displayed, all other
*       columns are hidden, to be used in  the student
*       information form.

##      INITTABLE #masterfrm studenttbl READ
##          (#SBDATE = CHAR(25),
##          #SGPA = FLOAT,
##          #SIDNO = INTEGER,
##          #SCOMMENT = CHAR(200),
##          #SADVISOR = CHAR(20))

##     DISPLAY #masterfrm UPDATE

##     INITIALIZE
##     {
##            MESSAGE "Enter an Advisor name . . ."
##            SLEEP 2
##     }

##     ACTIVATE MENUITEM "Students", FIELD "pname"
##     {

*           Load the students of the specified professor
##           GETFORM (PNAME = #pname)

*          If no professor name is given then resume
           IF PNAME = SPACES THEN
##                    RESUME FIELD #pname
           END-IF.

*          Verify the professor exists. Local error handling just
*          prints the message, and continues. We assume that each
*          professor has exactly one department.

           MOVE SPACES TO PDEPT.

##           RETRIEVE (PDEPT = p.#pdept, PNAME = p.#pname)
##                 WHERE p.#pname = PNAME

           IF PDEPT = SPACES THEN
                 MOVE SPACES TO MSGBUF
                 STRING "No professor with name """
                        DELIMITED BY SIZE,
                        PNAME DELIMITED BY " ",
                        """ [RETURN]" DELIMITED BY SIZE
                        INTO MSGBUF
```

```
##                        PROMPT NOECHO (MSGBUF, RESPBUF)
##                        RESUME FIELD #pname
           END-IF.

*          Fill the department field and load students
##          PUTFORM (#pdept = PDEPT, #pname = PNAME)

*          Refresh for query
##          REDISPLAY

          PERFORM LOAD-STUDENTS THROUGH END-LOAD.

##          RESUME FIELD studenttbl

##        }

##       ACTIVATE MENUITEM "Zoom"
##       {

*           Confirm that user is on "studenttbl", and that the
*           table field is not empty. Collect data from the
*           row and zoom for browsing and updating.

##          INQUIRE_FRS FIELD #masterfrm (ISTABLE = table)

          IF ISTABLE = 0 THEN
##              PROMPT NOECHO
##                     ("Select from the student table [RETURN]",
##                       RESPBUF)
##              RESUME FIELD studenttbl
          END-IF.

##        INQUIRE_FRS TABLE #masterfrm (LASTROW = lastrow)

          IF LASTROW = 0 THEN
##              PROMPT NOECHO ("There are no students [RETURN]",
##                             RESPBUF)
##              RESUME FIELD #pname
          END-IF.

*         Collect all data on student into global record
##        GETROW #masterfrm studenttbl
##              (SNAME = #sname,
##               SAGE = #sage,
##               SBDATE = #sbdate,
##               SGPA = #sgpa,
##               SIDNO = #sidno,
##               SCOMMENT = #scomment,
##               SADVISOR = #sadvisor)

*         Display "studentfrm", and if any changes were made make
*         the updates to the local table field row. Only make
*         updates to the columns corresponding to writable fields
*         in "studentfrm". If the student changed advisors, then
*         delete this row from the display.

          MOVE SADVISOR TO OLD-ADVISOR.

          PERFORM STUDENT-INFO-CHANGED THROUGH END-STUDENT.
```

```
        IF CHANGED-DATA = 1 THEN
               IF OLD-ADVISOR NOT = SADVISOR THEN
##                    DELETEROW #masterfrm studenttbl
               ELSE
##                    PUTROW #masterfrm studenttbl
##                           (#sgpa = SGPA,
##                            #scomment = SCOMMENT,
##                            #sadvisor = SADVISOR)
               END-IF
        END-IF.

##      }

##      ACTIVATE MENUITEM "Quit", FRSKEY2
##      {
##             BREAKDISPLAY
##      }
##      FINALIZE

        END-MASTER.
               EXIT.

**
*       Paragraph: LOAD-STUDENTS
*
*         For the current professor name, this paragraph loads
*         into the "studenttbl" table field all the students
*         whose advisor is the professor with that name.
**

        LOAD-STUDENTS.

##      MESSAGE "Retrieving Student Information . . ."
##      CLEAR FIELD studenttbl

##      RETRIEVE (SNAME = s.#sname,
##                SAGE = s.#sage,
##                SBDATE = s.#sbdate,
##                SGPA = s.#sgpa,
##                SIDNO = s.#sidno,
##                SCOMMENT = s.#scomment,
##                SADVISOR = s.#sadvisor)
##      WHERE s.#sadvisor = PNAME
##      {
##         LOADTABLE #masterfrm studenttbl
##                   (#sname = SNAME,
##                    #sage = SAGE,
##                    #sbdate = SBDATE,
##                    #sgpa = SGPA,
##                    #sidno = SIDNO,
##                    #scomment = SCOMMENT,
##                    #sadvisor = SADVISOR)
##      }

        END-LOAD.
          EXIT.

**
*    Paragraph: STUDENT-INFO-CHANGED
*
*         Allow the user to zoom into the details of a selected
*         student. Some of the data can be updated by the user.
*         If any updates were made, then reflect these back into
*         the database table. The paragraph records whether or not
*         changes were made via the CHANGED-DATA variable. **
```

```
      STUDENT-INFO-CHANGED.

*     Control ADDFORM to only initialize once

      IF LOADFORM = 0 THEN
##          MESSAGE "Loading Student form . . ."
* ##         ADDFORM studentfrm
             CALL "add_studentfrm"
             MOVE 1 TO LOADFORM
      END-IF.

##    DISPLAY #studentfrm FILL
##    INITIALIZE (#sname = SNAME,
##                #sage = SAGE,
##                #sbdate = SBDATE,
##                #sgpa = SGPA,
##                #sidno = SIDNO,
##                #scomment = SCOMMENT,
##                #sadvisor = SADVISOR)

##    ACTIVATE MENUITEM "Write", FRSKEY4
##    {

*         If changes were made then update the database table.
*         Only bother with the fields that are not read-only.

##        INQUIRE_FRS form (CHANGED-DATA = change)
          IF CHANGED-DATA = 0 THEN
##            BREAKDISPLAY
          END-IF.
##        VALIDATE
##        MESSAGE "Writing changes to database. . ."

##        GETFORM (SGPA = #sgpa,
##                 SCOMMENT = #scomment,
##                 SADVISOR = #sadvisor)

*         Enforce integrity of professor name.
          MOVE 0 TO VALID-ADVISOR
##        RETRIEVE (VALID-ADVISOR = 1)
##              WHERE p.#pname = SADVISOR

          IF VALID-ADVISOR = 0 THEN
##                MESSAGE "Not a valid advisor name"
##                SLEEP 2
##                RESUME FIELD #sadvisor
          ELSE
##                REPLACE s (#sgpa = SGPA, #scomment = SCOMMENT,
##                           #sadvisor = SADVISOR)
##                WHERE s.#sidno = SIDNO
##             BREAKDISPLAY
          END-IF.

##    }

##    ACTIVATE MENUITEM "End", FRSKEY3
##    {

*         Quit without submitting changes
          MOVE 0 TO CHANGED-DATA.
##        BREAKDISPLAY

##    }
##    FINALIZE
```

```
      END-STUDENT.
      EXIT.
```

VMS

The following two **create** statements describe the Professor and Student database tables.

```
##      CREATE student            /* Graduate student table */
##            (sname     = c25,       /* Name */
##             sage      = i1,        /* Age */
##             sbdate    = c25,       /* Birth date */
##             sgpa      = f4,        /* Grade point average */
##             sidno     = i4,        /* Unique student number */
##             scomment  = text(200), /* General comments */
##             sadvisor  = c25)       /* Advisor's name */

##        CREATE professor        /* Professor table */
##            (pname = c25,           /* Professor's name */
##             pdept = c10)           /* Department */

IDENTIFICATION DIVISION.
PROGRAM-ID. STUDENT-ADMINISTRATOR.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
##      DECLARE

* Global grad student record maps to database table
##       01 GRAD.
##           02 SNAME         PIC X(25).
##           02 SAGE          PIC S9(4) USAGE COMP.
##           02 SBDATE        PIC X(25).
##           02 SGPA          USAGE COMP-1.
##           02 SIDNO         PIC S9(9) USAGE COMP.
##           02 SCOMMENT      PIC X(200).
##           02 SADVISOR      PIC X(25).

* Professor info maps to database table
##       01 PROF.
##          02 PNAME          PIC X(25).
##          02 PDEPT          PIC X(10).

* Row number of last row in student table field
##       01 LASTROW           PIC S9(9) USAGE COMP.

* Is user on a table field?
##       01 ISTABLE           PIC S9 USAGE COMP.

* Were changes made to data in student form?
##       01 CHANGED           PIC S9 USAGE COMP.

* Did user enter a valid advisor name?
##       01 VALID-ADVISOR     PIC S9 USAGE COMP.

* Studentfrm loaded?
##       01 LOADFORM          PIC S9 USAGE COMP VALUE IS 0.

* Local utility buffers
##       01 MSGBUF            PIC X(100).
##       01 RESPBUF           PIC X.
##       01 OLD-ADVISOR       PIC X(25).
```

```
* Externally compiled forms
##    01 MASTERF    PIC S9(9) USAGE COMP VALUE EXTERNAL Masterfrm.
##    01 STUDENTF   PIC S9(9) USAGE COMP VALUE EXTERNAL Studentfrm.

**
* Procedure Division: STUDENT-ADMINISTRATOR
*
*        Start up program, Ingres, and the FORMS system and
*        call Master driver.
**

PROCEDURE DIVISION.
SBEGIN.

##       FORMS

##       MESSAGE "Initializing Student Administrator . . ."

##       INGRES "personnel"
##       RANGE OF p IS professor, s IS student

         PERFORM MASTER THROUGH END-MASTER.

##       CLEAR SCREEN
##       ENDFORMS
##       EXIT
         STOP RUN.

**
* Paragraph: MASTER
*
*        Drive the application, by running "masterfrm", and
*        allowing the user to "zoom" into a selected student.
**
MASTER.

##       ADDFORM MASTERF

* Initialize "studenttbl" with a data set in READ mode.
* Declare hidden columns for all the extra fields that the
* program will display when more information is requested about
* a student. Columns "sname" and "sage" are displayed, all
* other columns are hidden, to be used in the student information
* form.

##       INITTABLE #masterfrm studenttbl READ
##              (#SBDATE = CHAR(25),
##               #SGPA = FLOAT,
##               #SIDNO = INTEGER,
##               #SCOMMENT = CHAR(200),
##               #SADVISOR = CHAR(20))

##       DISPLAY #masterfrm UPDATE

##       INITIALIZE
##       {
##          MESSAGE "Enter an Advisor name . . ."
##               SLEEP 2
##       }

##       ACTIVATE MENUITEM "Students", FIELD "pname"
##       {

* Load the students of the specified professor

##          GETFORM (PNAME = #pname)
```

```
* If no professor name is given then resume

            IF PNAME = "" THEN
##                  RESUME FIELD #pname
            END-IF.

* Verify the professor exists. Local error handling just prints
* the message, and continues. We assume that each professor has
* exactly one department.

            MOVE "" TO PDEPT.

##          RETRIEVE (PDEPT = p.#pdept, PNAME = p.#pname)
##                  WHERE p.#pname = PNAME

            IF PDEPT = "" THEN
               MOVE "" TO MSGBUF
               STRING "No professor with name """ DELIMITED BY SIZE,
               PNAME DELIMITED BY " ",
               """ [RETURN]" DELIMITED BY SIZE
               INTO MSGBUF
##             PROMPT NOECHO (MSGBUF, RESPBUF)
##             CLEAR FIELD ALL
##             RESUME FIELD #pname
            END-IF.

* Fill the department field and load students

##          PUTFORM (#pdept = PDEPT, #pname = PNAME)

* Refresh for query

##          REDISPLAY

            PERFORM LOAD-STUDENTS THROUGH END-LOAD.

##          RESUME FIELD studenttbl

##       }

##       ACTIVATE MENUITEM "Zoom"
##       {

* Confirm that user is on "studenttbl", and that the table

* field is not empty. Collect data from the row and zoom
* for browsing and updating.

##       INQUIRE_FRS FIELD #masterfrm (ISTABLE = table)

         IF ISTABLE = 0 THEN
##             PROMPT NOECHO
                  ("Select from the student table [RETURN]",
##                  RESPBUF)
##             RESUME FIELD studenttbl
         END-IF.

##       INQUIRE_FRS TABLE #masterfrm (LASTROW = lastrow)

         IF LASTROW = 0 THEN

##             PROMPT NOECHO ("There are no students [RETURN]",
##                            RESPBUF)
##             RESUME FIELD #pname
```

```
        END-IF.

* Collect all data on student into global record

##      GETROW #masterfrm studenttbl
##             (SNAME         = #sname,
##              SAGE          = #sage,
##              SBDATE        = #sbdate,
##              SGPA          = #sgpa,
##              SIDNO         = #sidno,
##              SCOMMENT      = #scomment,
##              SADVISOR      = #sadvisor)

* Display "studentfrm", and if any changes were made make the
* updates to the local table field row. Only make updates to the
* columns corresponding to writable fields in "studentfrm". If
* the student changed advisors, then delete this row from
* the display.

        MOVE SADVISOR TO OLD-ADVISOR.

        PERFORM STUDENT-INFO-CHANGED THROUGH END-STUDENT.

        IF CHANGED = 1 THEN
              IF OLD-ADVISOR NOT = SADVISOR THEN
##                  DELETEROW #masterfrm studenttbl
              ELSE
##                  PUTROW #masterfrm studenttbl
##                         (#sgpa = SGPA,
##                          #scomment = SCOMMENT,
##                          #sadvisor = SADVISOR)
             END-IF
          END-IF.

##     }

##       ACTIVATE MENUITEM "Quit", FRSKEY2
##       {
##            BREAKDISPLAY
##       }
##       FINALIZE

END-MASTER.

**
* Paragraph: LOAD-STUDENTS
*
*     For the current professor name, this paragraph loads into the
*     "studenttbl" table field all the students whose advisor is
*     the professor with that name.
**

LOAD-STUDENTS.

##     MESSAGE "Retrieving Student Information . . ."
##     CLEAR FIELD studenttbl
```

```
##     RETRIEVE (
##                 SNAME      = s.#sname,
##                 SAGE       = s.#sage,
##                 SBDATE     = s.#sbdate,
##                 SGPA       = s.#sgpa,
##                 SIDNO      = s.#sidno,
##                 SCOMMENT   = s.#scomment,
##                 SADVISOR   = s.#sadvisor)
##     WHERE s.#sadvisor     = PNAME
##     {
##             LOADTABLE #masterfrm studenttbl
##                               (#sname        = SNAME,
##                                #sage         = SAGE,
##                                #sbdate       = SBDATE,
##                                #sgpa         = SGPA,
##                                #sidno        = SIDNO,
##                                #scomment     = SCOMMENT,
##                                #sadvisor     = SADVISOR)
##     }

END-LOAD.

**
* Paragraph: STUDENT-INFO-CHANGED
*
*        Allow the user to zoom into the details of a selected
*        student. Some of the data can be updated by the user.
*        If any updates were made, then reflect these back into
*        the database table. The paragraph records whether or not
*        changes were made via the CHANGED variable.
**

STUDENT-INFO-CHANGED.

* Control ADDFORM to only initialize once

        IF LOADFORM = 0 THEN
##                MESSAGE "Loading Student form . . ."
##                ADDFORM STUDENTF
                MOVE 1 TO LOADFORM
        END-IF.

##      DISPLAY #studentfrm FILL
##      INITIALIZE
##                     (#sname      = SNAME,
##                      #sage       = SAGE,
##                      #sbdate     = SBDATE,
##                      #sgpa       = SGPA,
##                      #sidno      = SIDNO,
##                      #scomment   = SCOMMENT,
##                      #sadvisor   = SADVISOR)

##      ACTIVATE MENUITEM "Write", FRSKEY4
##      {

* If changes were made then update the database table. Only
* bother with the fields that are not read-only.

##                      INQUIRE_FRS form (CHANGED = change)

                        IF CHANGED = 0 THEN
##                            BREAKDISPLAY
                        END-IF.

##                      VALIDATE
##                      MESSAGE "Writing changes to database. . ."
```

```
##                      GETFORM
##                               (SGPA = #sgpa,
##                               SCOMMENT = #scomment,
##                               SADVISOR = #sadvisor)

* Enforce integrity of professor name.

                       MOVE 0 TO VALID-ADVISOR
##                     RETRIEVE (VALID-ADVISOR = 1)
##                        WHERE p.#pname = SADVISOR

                       IF VALID-ADVISOR = 0 THEN
##                        MESSAGE "Not a valid advisor name"
##                        SLEEP 2
##                        RESUME FIELD #sadvisor
                       ELSE
##                    REPLACE s (#sgpa = SGPA, #scomment = SCOMMENT,
##                                  #sadvisor = SADVISOR)
##                                  WHERE s.#sidno = SIDNO
##                      BREAKDISPLAY
                      END-IF.

##      }

##      ACTIVATE MENUITEM "End", FRSKEY3
##      {

* Quit without submitting changes

                MOVE 0 TO CHANGED.
##              BREAKDISPLAY

##      }
##      FINALIZE

END-STUDENT.
```

Chapter 4: Embedded QUEL for Fortran

This chapter describes the use of EQUEL with the Fortran programming language.

EQUEL Statement Syntax for Fortran

This section describes the language-specific ground rules for embedding QUEL database and forms statements in a Fortran program. An EQUEL statement has the following general syntax:

*EQUEL_statement*

For information on QUEL statements, see the *QUEL Reference Guide*. For information on EQUEL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe how to use the various syntactical elements of EQUEL statements as implemented in Fortran.

Margin

There are no specified margins for EQUEL statements in Fortran. Always place the two number signs (##) in the first two positions of a line. The rest of the statement can begin anywhere else on the line. In the preprocessor, the statement field follows the first tab.

Terminator

An EQUEL/Fortran statement does not need a statement terminator. Although the convention is to not use a statement terminator in EQUEL statements, the preprocessor does allow a semicolon at the end of EQUEL statements. The preprocessor also ignores it.

For example, it interprets the following two statements as the same:

```
##   sleep 1
```

and

```
##   sleep 1;
```

EQUEL statements that are made up of a few other statements, such as a **display** loop, only allow a semicolon after the last statement. For example:

```
##   display empfrm
##   initialize
##   activate menuitem "Help"
##   {
##        message "No help yet";
##        sleep 2;
##   }
##   finalize;
```

When using a **retrieve** loop, place a semicolon after the **retrieve** statement to disassociate the loop code inside the braces from the **retrieve** statement itself. Variable declarations made visible to EQUEL follow the normal Fortran declaration syntax. Therefore, do not use a statement terminator on variable declarations.

Line Continuation

There are no special line-continuation rules for EQUEL/Fortran. You can break an EQUEL statement between words and continue it on any number of subsequent lines. An exception to this rule is that you cannot continue a statement between two words that are reserved when they appear together, such as **declare cursor**. For a list of double keywords, see the *QUEL Reference Guide*. Start each continuation line with ## characters. No continuation indicator is necessary. You can put blank lines between continuation lines. For example, the following **retrieve** statement is continued over two lines:

```
## retrieve (empnam = e.ename)
##        where e.eno = enum
```

As a result, the preprocessor output includes a Fortran continuation indicator on any continued lines.

If you want to continue a character-string constant across two lines, end the first line with a backslash character (\) and continue the string at the beginning of the next line. In this case, do not place ## characters at the beginning of the continuation lines.

For examples of string continuation, see String Literals in this chapter.

Comments

Two kinds of comments can appear in an EQUEL program: EQUEL comments and host language comments. Use the /* and */ characters to delimit EQUEL comments. These characters must appear on lines beginning with the ## sign. For example:

```
## /* Update name and salary*/
## append to employee (ename = empnam, esal = esal*.1)
```

The preprocessor strips EQUEL comments out of the program that appear on lines beginning with the ## sign. These comments do not appear in the output file.

The capitol C delimits Fortran host language comments. These comments must start on a separate line. For example:

```
##   message "No permission . . ."
C    No user access
```

The preprocessor treats host language comments that appear on lines that do not begin with the ## sign as host code and passes them through to the output file unchanged. Therefore, if you want source code comments in the preprocessor output, enter them as Fortran comments on lines that are not EQUEL lines.

The following restrictions apply to any EQUEL or Fortran comments in an EQUEL/Fortran program:

- If anything other than ## appears in the first two positions of a line of EQUEL source, the precompiler treats the line as host code and ignores it. The only exception to this is a string-continuation line. See String Literals in this chapter.
- Comments cannot appear in string constants. If this occurs, the preprocessor interprets the intended comment as part of the string constant.
- In general, EQUEL comments can be put in EQUEL statements wherever a space can legally occur. However, comments cannot appear between two words that are reserved when they appear together, such as **declare cursor**. See the list of EQUEL reserved words in the *QUEL Reference Guide.*

VMS

In VMS, you can also use the ! character instead of C to delimit Fortran host language comment which extends to the end of the line. It can appear on a line beginning with the ## sign. For example:

```
## message "No permission . . ." !No user access
```

Windows

In Windows, you can also use the ! character instead of C or c to delimit Fortran host language comment that extends to the end of the line. It can appear on a line beginning with the ## sign. For example:

```
## message "No permission . . ." !No user access
```

String Literals

You can use either double quotes or single quotes to delimit string literals in EQUEL/Fortran. Be sure that you begin and end the string with the same delimiter.

Whichever quote mark you use, you can embed it as part of the literal itself by doubling it. For example:

```
##   append comments
##   (field1 = "a double "" quote is in this string")
```

or

```
##   append comments
##   (field1 = 'a single '' quote is in this string')
```

To continue an EQUEL statement to additional lines, use the backslash character (\) at the end of the first line. Any leading spaces on the next line are considered part of the string. Therefore, the continued string should start in column 1, where a statement label would normally appear in non-EQUEL lines. For example, the following are legal EQUEL statements:

```
##   message 'Please correct errors found in updating\
the database tables.'

##   append to employee (empnam = "Freddie \
Mac", empnum = 222)
```

Fortran Variables and Data Types

This section describes how to declare and use Fortran program variables in EQUEL.

Variable and Type Declarations

This section describes how to declare variables to EQUEL. It provides a general description of declaration sections and a detailed description of the declaration syntax for all data types.

EQUEL Variable Declaration Procedures

Any Fortran language variable an EQUEL statement uses must be made known to the preprocessor so that it can determine the type of the variable. EQUEL/Fortran does not know the implicit typing conventions Fortran uses, so you must explicitly declare all variables. The preprocessor uses the declaration to set up type information for the Ingres runtime system.

Use two number signs (##) to begin a declaration of a variable in an EQUEL/Fortran program. Begin the signs in the first column position of the line. If the EQUEL statement does not use a variable, you do not need to use number signs.

The Declare and Declare Forms Statements

Prior to any EQUEL declarations or statements in a program unit, you must issue the following statement:

```
## declare
```

This statement must follow all *implicit* statements in the unit. If there are no *implicit* statements, the ## **declare** directive must be the first statement in the unit. When your program unit includes EQUEL/FORMS statements, you must use a slightly different variant of the ## **declare** directive:

```
## declare forms
```

These statements make the preprocessor generate a Fortran **include** statement that includes a file of declarations the Ingres runtime system needs. You cannot link an EQUEL/Fortran program unless you include one of these statements in every program unit that contains EQUEL statements.

The **declare** statements also served the purpose of scope delimiters in earlier versions of EQUEL/Fortran. For examples of string continuation, see The Scope of Variables in this chapter.

Reserved Words in Declarations

In declarations, all EQUEL keywords are reserved. Therefore, you cannot declare types or variables with the same name as those keywords. Also, when you use the following EQUEL/Fortran keywords in declarations, they are reserved by the preprocessor and you cannot use them elsewhere, except in quoted string constants:

byte	**double**	**logical**	**program**	**structure**
character	**external**	**map**	**real**	**union**
complex	**function**	**parameter**	**record**	
declare	**integer**	**precision**	**subroutine**	

The EQUEL preprocessor does not distinguish between uppercase and lowercase in keywords. When it generates Fortran code, it converts any uppercase letters in keywords to lowercase. This rule is true only for keywords. The preprocessor does distinguish between case in program-defined types and variable names.

Variable and type names must be legal Fortran identifiers that begin with an alphabetic character.

Typed Data Declarations

UNIX

The preprocessor recognizes numeric variables declared with the following format:

data_type [*****default_type_len]
var_name [*****type_len] [**(**array_spec**)**]
{**,** *var_name* [*****type_len] [**(**array_spec**)**]

The preprocessor recognizes character variables declared with the following format:

data_type [*****default_type_len[**,**]]
var_name [**(**array_spec**)**] [*****type_len]
{, *var_name* [**(**array_spec**)**] [*****type_len]}

VMS

The preprocessor recognizes numeric variables declared with the following format:

data_type [*****default_type_len]
var_name [*****type_len] [**(**array_spec**)**] **[**/init_clause/]
{**,** *var_name* [*****type_len] [**(**array_spec**)**] [/init_clause/] }

The preprocessor recognizes character variables declared with the following format:

data_type [*****default_type_len[**,**]]
var_name [**(**array_spec**)**] [*****type_len] **[**/init_clause/]
{**,** *var_name* [**(**array_spec**)**] [*****type_len] **[**/init_clause/] }

Windows

The preprocessor recognizes numeric variables declared with the following format:

data_type [*****default_type_len]
var_name [*****type_len] [**(**array_spec**)**] **[**/init_clause/]
{**,** *var_name* [*****type_len] [**(**array_spec**)**] [/init_clause/] }

The preprocessor recognizes character variables declared with the following format:

data_type [*****default_type_len[**,**]]
var_name [**(**array_spec**)**] [*****type_len] **[**/init_clause/]
{**,** *var_name* [**(**array_spec**)**] [*****type_len] **[**/init_clause/] }

Syntax Notes:

- For information on the allowable *data_types*, see Data Types in this chapter.
- The *default_type_len* specifies the size of the declared variable. To specify size for a numeric type variable, use an integer literal of an acceptable length for the particular data type. To specify size for a **character** type variable, use an integer literal or a parenthesized expression, followed optionally by a comma. The preprocessor does not interpret the length field for variables of type **character** but merely passes that information to the output file. Note the default type lengths in the following declarations:

```
C  Declares "eage" a 2-byte integer
     integer*2           eage
C  Declares "stat" a 2-byte integer
     logical*2           stat
C  Declares "ename" a character string
     character*(4+len)   ename
```

- The *type_len* allows you to declare a variable with a length different from *default_type_len*. Again, you can use a parenthesized expression only to declare the length of character variable declarations. The type length for a numeric variable must be an integer literal that represents an acceptable numeric size. For example:

```
C  Default-sized integer and 2-byte integer
     integer         length
     integer*2       height
     character*15    name, socsec*(numlen)
```

  Some Fortran compilers do not permit the redeclaration of the length of a character variable.

- The variable names must be legal Fortran identifiers.
- The *array_spec* must conform to Fortran syntax rules. The preprocessor simply notes that the declared variable is an array, but does not parse the *array_spec* clause.

VMS

Note that, if you specify both an array and a type length, the order of those two clauses differs depending on whether the variable being declared is of character or numeric type. Note the following examples of array declarations:

```
## character*16 enames(100), edepts(15)*10
                ! Array specification first

## real*4 salestab(5,12), yeartotals*8(12)
                ! Type length first
```

- The preprocessor allows you to initialize a variable or array in the declaration statement using the *init_clause*. The preprocessor accepts, but does not examine, any initial data. The Fortran compiler, however, later detects any errors in the initial data. For example:

```
## real*8 initcash /512.56/
## character*4 baseyear /'1950'/
```

```
## character*4 year /1950/
   ! Acceptable to preprocessor but not to compiler
```

Do not continue initial data over multiple lines. If an initialization value is too long for the line, as could be the case with a string constant, use the Fortran **data** statement instead.

Constant Declarations

To declare constants to EQUEL/Fortran, use the Fortran **parameter** statement with the following syntax:

UNIX

parameter (const_name **=** value {**,** const_name **=** value}**)**

VMS

parameter const_name **=** value {**,** const_name **=** value}

Windows

parameter const_name **=** value {**,** const_name **=** value}
or
parameter(const_name **=** value {**,** const_name **=** value}**)**

Syntax Notes:

- The data type of *const_name* derives its data type from the data type of *value*. Do not put explicit data type declarations in **parameter** statements. In addition, as with variable declarations, the preprocessor does not assign a data type based on the first letter of *const_name*.
- The *value* can be a **real**, **integer** or **character** literal. It cannot be an expression or a symbolic name.

The following example declarations illustrate the **parameter** statement:

UNIX

```
C real constant
parameter (pi = 3.14159 )
C integer and real
parameter (bigint = 2147483648, bgreal = 999999.99)
```

VMS

```
##  parameter pi = 3.14159   ! real constant
##  parameter bigint = 2147483648, bgreal = 999999.99
            ! integer and real
```

Windows

```
##  parameter pi = 3.14159   ! real constant
##  parameter(bigint = 2147483648, bgreal = 999999.99)
            ! integer and real
```

Data Types

The EQUEL/Fortran preprocessor accepts the elementary Fortran data types shown in the following table. The table maps these types to corresponding Ingres types. For more information on type mapping between Ingres and Fortran data, see Data Type Conversion in this chapter.

Fortran Data Type	Ingres Types
integer	**integer**
integer*N* **where** *N* = 2 or 4	**integer**
logical	**integer**
logical*N* where *N* = 1, 2 or 4	**integer**
byte	**integer**
real	**float**
real*N* where *N* = 4 or 8	**float**
double precision	**float**
character*N* where *N* > 0	**character**
real*8	**decimal**

The Integer Data Type

The Fortran compiler allows the default size of **integer** variables to be either two or four bytes in length, depending on whether the **-i2** compiler flag (UNIX), the **noi4** qualifier (VMS), or the /integer_size:16 compiler option (Windows) is set.

EQUEL/Fortran also supports this feature by means of the **-i2** preprocessor flag. This flag tells the preprocessor to treat the default size of **integer** variables as two instead of the normal default size of four bytes. For more information on type mapping between Ingres and Fortran data, see Precompiling, Compiling, and Linking an EQUEL Program in this chapter.

You can explicitly override the default size when declaring the Fortran variable to the preprocessor. To do so, you must specify a size indicator (*2 or *4) following the **integer** keyword, as these examples illustrate:

```
integer*4       bigint
integer*2       smalli
```

The preprocessor then treats these variables as a four-byte integer and a two-byte integer, regardless of the default setting.

UNIX

The preprocessor treats the **logical** data type as an **integer** data type. A **logical** variable has a default size of 4 bytes. To override this default size, use a size indicator of 2 or 4. For example:

```
logical*2 log2
logical*4 log4
logical*1 log1
```

VMS

The preprocessor treats **byte** and **logical** data types as **integer** data types. A **logical** variable has a default size of either two or four bytes, according to whether the **-i2** flag has been set. You can override this default size by a size indicator of 1, 2 or 4. For example:

```
## logical log1*1, log2*2, log4*4
```

Windows

The preprocessor treats **byte** and **logical** data types as an **integer** data type. A **logical** variable has a default size of 4 bytes. To override this default size, use a size indicator of 2 or 4. For example:

```
## logical*2 log2
## logical*4 log4
## logical*1 log1
```

The byte data type has a size of one **byte**. You cannot override this size.

You can use an **integer** or **byte** variable with any numeric-valued object to assign or receive numeric data. For example, you can use such a variable to set a field in a form or to select a column from a database table. This variable can also specify simple numeric objects, such as table field row numbers. You can use a **logical** variable to assign or receive integer data, although your program must restrict its value to 1 and 0, which map respectively to the Fortran logical values **.TRUE.** and **.FALSE.**.

The Real Data Type

EQUEL/Fortran accepts **real** and **double precision** as legal real data types. The preprocessor accepts both 4-byte and 8-byte **real** variables. It makes no distinction between an 8-byte **real** variable and a **double precision** variable. The default size of a **real** variable is 4 bytes. However, you can override this size if you use a size indicator (*8) that follows the **real** keyword or the variable's name.

You can only use a real variable to assign or receive numeric data (both real, decimal, and integer). You cannot use it to specify numeric objects, such as table field row numbers.

VMS

The preprocessor expects the internal format of **real** and **double** precision variables to be the standard VAX format. For this reason, you should not compile your program with the **g_floating** qualifier.

```
C 4-byte real variable
##    real salary
C 8-byte real variable
##    real*8 yrtoda
C 8-byte real variable
##    double precision saltot
##    real salary, yrtodate*8
```

Only use a **real** variable to assign or receive numeric data (both **real** and **integer**). Do not use it to specify numeric objects, such as table field row numbers.

The Character Data Type

Fortran variables of type **character** are compatible with all Ingres character string objects. EQUEL/Fortran does not need to know the declared length of a character string variable to use it at runtime. Therefore, it does not check the validity of any expression or symbolic name that declares the length of the string variable. You should ensure that your string variables are long enough to accommodate any possible runtime values. For information on the interaction between character string variables and Ingres data at runtime, see Runtime Character Conversion in this chapter.

```
## character*7 first
## character*10 last
## character*1 init
## character*(bufsiz) msgbuf
```

Structure and Record Declarations

EQUEL/Fortran supports the declaration and use of user-defined structure variables. The syntax of a structure definition is:

structure [**/***structdef_name***/**] [*field_namelist*]
field_declaration
{*field_declaration*}
end structure

Syntax Notes:

- The *structdef_name* is optional only for a nested structure definition.
- The *field_namelist* is allowed only with a nested structure definition. Each name in the *field_namelist* constitutes a field in the enclosing structure.
- The *field_declaration* can be a typed data declaration (see Typed Data Declarations in this chapter), a nested structure declaration, a **union** declaration, a **record** declaration, or a **fill** item.

The syntax of a **union** declaration is as follows:

union
map_declaration
map_declaration
{*map_declaration*}
end union

where *map_declaration* is:

map
field_declaration
{*field_declaration*}
end map

- Only *field_declarations* that are referenced in EQUEL statements need to be declared to EQUEL. The following example declares a Fortran structure with a member "checked" that is not known to EQUEL.

```
## structure /address/
## integer number
## character*20 street
## character*10 town
## integer*2 zip
 logical checked
## end structure
## record /address/addr
```

To use a structure with EQUEL statements, you must both define the structure and declare the structure's record to EQUEL. The **record** declaration has the following syntax:

record /*structdef_name***/** *structurename* {**,**[**/***structdef_name***/**] *structurename*}

Syntax Note:

The *structdef_name* must be previously defined in a **structure** statement.

For information on the use of structure variables in EQUEL statements, see Using a Structure Member in this chapter.

The following example includes a structure definition and a record declaration:

```
## structure /name_map/
## union
## map
##  character*30 fullname
## end map
## map
##  character*10 firstnm
##  character*2 init
##  character*18 lastnm
## end map
## end union
## end structure
```

```
## record /name_map/ empname
```

The next example shows the definition of a structure containing an array of nested structures:

```
## structure /class_struct/
## character*10 subject
## integer*2 year
## structure student(100)
C No structure definition name needed
## character*12 name
## byte grade
## end structure
## end structure

## record /class_struct/ classrec
```

Indicator Variables

An *indicator variable* is a 2-byte integer variable. There are three ways to use in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a null.
- In a statement that sets Ingres data, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable was large enough to hold the full length of the returned character string.

The following declaration illustrates how to declare a null indicator variable:

```
C Indicator variable
## integer*2 ind
```

Assembling and Declaring External Compiled Forms - VMS

You can precompile your forms in the Visual Forms Editor (VIFRED). By doing so, you save the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file in which to write the MACRO description. After the file is created, you can use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the EQUEL/FORMS statement **addform** can refer to this global object, you must declare it to EQUEL with the following syntax:

integer *formname*

Next, in order for the program to access the external form definition, you must declare the *formname* as an external symbol:

external *formname*

This second declaration is not an EQUEL declaration and you should not precede it by the ## mark. Its purpose is to inform the linker to associate the global symbol in the compiled form file with the object of the **addform** statement.

Syntax Notes:

- *formname* is the actual name of the form; it appears as the title of the form in EQUEL/FORMS statements other than the **addform** statement. It is also the name that VIFRED gives to the global object in the compiled form file. In all EQUEL/FORMS statements other than the **addform** statement that expect a form name, you must dereference *formname* with # so that it is interpreted as a name and not as an integer variable.
- The EXTERNAL statement associates the external form definition with the integer object used by the **addform** statement.

The following example illustrates these points:

```
## integer empfrm
 external empfrm

## addform empfrm ! The global object
## display #empfrm ! The name of the form must be dereferenced
## ! because it is also the name of a variable
```

Compiling and Declaring External Compiled Forms - UNIX

You can precompile your forms in VIFRED. This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in C. VIFRED prompts you for the name of the file with the description. After the file is created, you can use the following **cc** command to compile it into linkable object code:

cc -c *filename*

This command produces an object file that contains a global symbol with the same name as your form.

Before the EQUEL/FORMS statement **addform** can refer to this global object, use the following syntax to declare it to EQUEL:

extern int ****formname*;**

Next, for the program to access the external form definition, you must declare the *formname* as an external symbol:

external *formname*

Because this second declaration is not an EQUEL declaration, do not precede it with the ## mark. Its purpose is to inform the linker to associate the global symbol in the compiled form file with the object of the **addform** statement.

Syntax Notes:

- *formname* is the actual name of the form and appears as the title of the form in EQUEL/FORMS statements other than the **addform** statement. It is also the name that VIFRED gives to the global object in the compiled form file.

  In all EQUEL/FORMS statements other than the **addform** statement that expect a form name you must dereference *formname* with # so that it is interpreted as a name and not as an integer variable.

- The EXTERNAL statement associates the external form definition with the integer object used by the **addform** statement.

```
##  integer empfrm
    external empfrm

C  The global object
##   ADDFORM empfrm
C  The name of the form must be dereferenced
C  because it is also the name of a variable
## DISPLAY #empfrm
```

Compiling and Declaring External Compiled Forms - Windows

You can precompile your forms in VIFRED. By doing so, you save the time otherwise required at run time to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in C. VIFRED prompts you for the name of the file in which to write the description. After the file is created, you can use the following Windows command to compile it into a linkable object module:

cl –c *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the EQUEL/FORMS statement **addform** can refer to this global object, you must declare it to EQUEL with the following syntax:

integer *formname*

Next, in order for the program to access the external form definition, you must declare the *formname* as an external symbol:

external *formname*

This second declaration is not an EQUEL declaration and you should not precede it by the ## mark. Its purpose is to inform the linker to associate the global symbol in the compiled form file with the object of the **addform** statement.

Syntax Notes:

- *formname* is the actual name of the form; it appears as the title of the form in EQUEL/FORMS statements other than the **addform** statement. It is also the name that VIFRED gives to the global object in the compiled form file. In all EQUEL/FORMS statements other than the **addform** statement that expect a form name, you must dereference *formname* with # so that it is interpreted as a name and not as an integer variable.
- The EXTERNAL statement associates the external form definition with the integer object used by the **addform** statement.

The following example illustrates these points:

```
## integer empfrm
 external empfrm

## addform empfrm ! The global object
## display #empfrm ! The name of the form must be dereferenced
## ! because it is also the name of a variable
```

Concluding Example

The following example contains some simple EQUEL/Fortran declarations:

UNIX

```
##    declare
C     Variables of each data type
##    byte           dbyte
##    logical*4      log4
##    logical        logdef
##    integer*2      dint2
##    integer*4      dint4
##    integer        intdef
##    real*4         dreal4
##    real*8         dreal8
##    real           dreal
```

```
C     Constant
##    parameter  (MAXVAL = 1000)

##    character*12   dbname
##    character*12   drmnam, tblnam, colnam

C     Compiled forms
##    integer        empfrm, dptfrm
      external       empfrm,        dptfrm
```

VMS

```
##    declare
##    byte             d_byte !Variables of each data type
##    logical*1        d_log1
##    logical*2        d_log2
##    logical*4        d_log4
##    logical          d_logdef
##    integer*2        d_int2
##    integer*4        d_int4
##    integer          d_intdef
##    real*4           d_real4
##    real*8           d_real8
##    real             d_realdef
##    double precision d_doub

##    parameter MAX_PERSONS = 1000 ! Constant

##    character*12     dbname/'personnel'/
##    character*12     formname, tablename, columnname

##    structure     /person/ ! Structure with a union
##    byte       age
##    integer    flags
##    union
##            map
##                character*30 full_name
##            end map
##            map
##                character*12 firstname
##                character*18 lastname
##            end map
##    end union
##    end structure
                ! Record/array of records

##    record /person/ person, p_table(MAX_PERSONS)

##    integer  empfrm, deptform ! Compiled forms
      external empfrm, deptform ! Compiled forms
```

Windows

```
##    declare
##    byte             d_byte !Variables of each data type
##    logical*1        d_log1
##    logical*2        d_log2
##    logical*4        d_log4
##    logical          d_logdef
##    integer*2        d_int2
##    integer*4        d_int4
##    integer          d_intdef
##    real*4           d_real4
##    real*8           d_real8
```

```
##    real             d_realdef
##    double precision d_doub

##    parameter MAX_PERSONS = 1000 ! Constant

##    character*12     dbname/'personnel'/
##    character*12     formname, tablename, columnname

##    structure     /person/ ! Structure with a union
##    byte       age
##    integer    flags
##    union
##            map
##                character*30 full_name
##            end map
##            map
##                character*12 firstname
##                character*18 lastname
##            end map
##    end union
##    end structure
                ! Record/array of records

##    record /person/ person, p_table(MAX_PERSONS)

##    integer  empfrm, deptform ! Compiled forms
      external empfrm, deptform ! Compiled forms
```

The Scope of Variables

Variable names must be unique in their scope. The EQUEL/Fortran preprocessor understands scoping of variables if your program adheres to the following rules:

- To declare a scope for a program or subprogram, use the ## signal on the **program**, **subroutine** or **function** statement line and also on the line where the matching **end** statement appears. EQUEL considers the scope of variables declared in such a program or subprogram to be exactly that program unit. The variables can be local variables, common variables or subprogram dummy arguments (formal parameters).
- Be aware that without scoping information, the preprocessor considers the **declare** and **declare forms** statements to signal the closing of the previous scope and the opening of a new one. In other words, if your program has *not* used the ## signal on a **program**, **subroutine** or **function** statement, a **declare** statement begins a new scope. For a discussion of the EQUEL/Fortran **declare** statement, see The Declare and Declare Forms Statements in this chapter.

The following program fragments illustrate the scope of variables in an EQUEL/Fortran program:

```
## program emp
## declare

C The following two declarations will be visible to the
C preprocessor until the end of program 'emp'.
```

```
## integer empid
## real empsal
## real raise

C EQUEL statements using 'empid', 'empsal' and 'raise'

       call prcemp (empid)
       call prcsal (empsal, raise)
## end

## subroutine prcemp (empid)
## declare

C 'empid' must be redeclared to EQUEL because of new
C  scope

## integer empid

C EQUEL statements using 'empid'

## end

## subroutine prcsal (esal, raise)
## declare

C Declare only those formal parameters to EQUEL that
C will be used in EQUEL statements.

## real esal

C EQUEL statements using 'esal'

## end
```

Variable Usage

Fortran variables declared to EQUEL can substitute for most elements of EQUEL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. The generic uses of host language variables in EQUEL statements are discussed further in the *QUEL Reference Guide.* The following discussion covers only the usage issues particular to Fortran language variable types.

You must verify that the statement using the variable is in the scope of the variable's declaration. For a discussion of variables in an EQUEL/Fortran program, see The Scope of Variables in this chapter. As an example, the following **retrieve** statement uses the variables "namvar" and "numvar" to receive data, and the variable "idno" as an expression in the **where** clause:

```
## retrieve (namvar = employee.empname,
##         numvar = employee.empnum) where
##         employee.empnum = idno
```

Simple Variables

The following syntax refers to a simple scalar-valued variable (integer, floating-point, or character string) :

simplename

Syntax Notes:

- If you use the variable to send values to Ingres, the variable can be any scalar-valued variable.
- If you use the variable to receive values from Ingres, the variable can only be a scalar-valued variable.

The following example shows a message handling routine. It passes two scalar-valued variables as parameters: buffer, which is a character string, and secs, which is an integer variable.

```
## subroutine PrtMsg(buffer, secs)
## declare forms
## character*(*) buffer
## integer secs

## message buffer
## sleep secs

## end
```

Array Variables

The following syntax refers to an array variable:

arrayname (*subscripts*)

Syntax Notes:

- Subscript the variable because only scalar-valued elements (integers, floating-point, and character strings) are legal EQUEL values.
- The EQUEL preprocessor does not evaluate subscript values when the array is declared and referenced. Consequently, even though the preprocessor confirms that array subscripts have been used, it accepts illegal subscript values. You must make sure that the subscript is legal. For example, the preprocessor accepts both of the following references, even though only the first is correct:

```
## real salary(5)
C  declaration

## APPEND TO employee (esal = salary(1))
C   Correct reference
## APPEND TO employee (esal = salary(-1))
C  Incorrect reference
```

- Do not subscript arrays of variable addresses that are used with **param** target lists. For example:

```
## character*200 target
## integer*4 addr(10)

C Array of variable addresses

## RETRIEVE (PARAM (target, addr))
```

For more information about parameterized target lists, see Dynamically Built Param Statements in this chapter.

The following example uses the variable "i" as a subscript. However, the variable does not need to be declared to EQUEL because array subscripts are not parsed or evaluated.

UNIX

```
## character*8  frnams(3)
   integer        i

   data         frnams /'empfrm', 'dptfrm'. 'hlpfrm'/

   do 10 i = 1, 3
##    FORMINT frnams(i)
10    continue 1
```

VMS

```
##  character*8 formnames(3) / 'empfrm', 'deptform',
'helpform'/
integer i

do i=1,3
## Forminit formnames(i)
end do
```

Windows

```
##    declare
##    character*8 formnames(3)/'empfrm','deptform','helpform'/
##    declare forms
       integer i
       character*(*) active
       do i=1,3
       active = formname(i)
##    Forminit active
       end do
```

Structure Variables - VMS only

You cannot use a structure variable as a single entity. Only elementary structure members can communicate with Ingres data. This member must be a scalar value (integer, floating-point, or character string).

Using a Structure Member

The syntax EQUEL uses to refer to a structure member is the same as in Fortran:

structure.*member*{.*member*}

Syntax Notes:

- The structure member the above reference denotes must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and structures, but the last object referenced must be a scalar value. Thus, the following references are all legal in an EQUEL statement, assuming they all translate to scalar values:

```
      employee.sal
C     Member of a structure
      person(3).name
C     Member of an element of an array
      structure.mem2.mem3.age
C     Deeply nested member
```

- In general, the preprocessor supports unambiguous and fully qualified structure member references.

Using Indicator Variables

The syntax for referring to an indicator variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

host_variable:indicator_variable

Syntax Notes:

- The indicator variable can be a simple variable, an array element or a structure member that yields a short integer. For example:

```
## integer*2    indvar, indarr(5)
     var_1:indvar
     var_3:indarr(2)
```

Data Type Conversion

A Fortran variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables. Ingres character values can be set by and retrieved into character variables.

Data type conversion occurs automatically for different numeric types, such as from floating-point Ingres database column values into integer Fortran variables, and for character strings, such as from varying-length Ingres character fields into fixed-length Fortran character string buffers.

Ingres does *not* automatically convert between numeric and character types. You must use one of the Ingres type conversion functions or a Fortran conversion routine for this purpose.

The following table shows the specific type correspondences for each Ingres data type.

Ingres and Fortran Data Type Compatibility

Ingres Type	Fortran Type
cN	**character*N**
text*(N)*	**character*N**
char*(N)*	**character*N**
varchar*(N)*	**character*N**
i1	**byte**
i2	**integer*2**
i4	**integer*4**
f4	**real*4**
f8	**real*8**
date	**character*25**
money	**real*8**

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and the forms system and numeric Fortran variables. It follows the standard type conversion rules. For example, if you assign a **real** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion when assigning Ingres numeric values to Fortran variables.

The default size of integers in EQUEL/Fortran is four bytes. You can change the default size to two bytes by means of the **-i2** preprocessor flag. If you use this flag, you must compile the program with the **-i2** flag for UNIX, the **noi4** qualifier for VMS, or the /integer_size:16 flag for Windows.

The Ingres **money** type is represented as an 8-byte real value, compatible with a Fortran **real*8**.

Runtime Character Conversion

Automatic conversion occurs between Ingres character string values and Fortran character variables. There are four string-valued Ingres objects that can interact with character variables:

- Ingres names, such as form and column names
- Database columns of type **c** or **char**
- Database columns of type **text** or **varchar**
- Form fields of type **c**

Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of Fortran character string variables that represent Ingres object names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string literals "empfrm " and "empfrm" refer to the same form, and "employees " and "employees" refer to the same database table.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **c** or **char**, a database column of type **text** or **varchar**, or a character-type form field. Ingres pads columns of type **c** and **char** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **text** or **varchar** or in form fields.

The Fortran convention is to blank-pad fixed-length character strings. For example, the character string "abc" is stored in a Fortran **character*5** variable as the string "abc " followed by two blanks.

When character data is retrieved from a database column or form field into a Fortran character variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data.

When inserting character data into an Ingres database column or form field from a Fortran variable, note the following conventions:

- When data is inserted from a Fortran variable into a database column of type **c** or **char** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a Fortran variable into a database column of type **text** or **varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **text** or **varchar** column. For example, when a string "abc" stored in a Fortran **character*5** variable as "abc " (refer to above) is inserted into the **text** or **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the Ingres **notrim** function. It has the following syntax:

  notrim(*charvar*)

  where *charvar* is a character variable. An example that demonstrates this feature follows later. If the **text** or **varchar** column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a Fortran variable into a **c** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

When comparing character data in a Ingres database column with character data in a Fortran variable, note the following convention:

- When comparing data in **c**, **character**, or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Trailing blanks are significant in **text**. Initial and embedded blanks are significant in **character**, **text**, and **varchar**; they are ignored in **c**.

Caution! As just described, the conversion of character string data between Ingres objects and Fortran variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion.

The Ingres **date** data type is represented as a 25-byte character string.

The following program fragment demonstrates the **notrim** function and the truncation rules explained above.

```
C   Program to illustrate significance of trailing

C   blanks in   TEXT datatype

    program txttype

##  declare
##  integer*2 row
##  character*7 data

C   data will have 'abc' followed by 4 blanks

    data = 'abc  '

##  ingres testdb

C   set up the table for testing

##  create texttype (#row = i2, #data = text(10))

C   The first APPEND adds the string 'abc' (blanks
C   truncated)
##  append to texttype (#row = 1, #data = data)

C   The second APPEND adds the string 'abc ', with
C   4 trailing  blanks

##  append to texttype (#row = 2, #data = NOTRIM(data))

C   The RETRIEVE will get the second row because the
C   NOTRIM   function in the previous APPEND caused
C   trailing blanks to be inserted as data.

##  retrieve (row = texttype.#row)

##  where length(texttype.#data) = 7

    print *, 'Row found = ', row
##  destroy texttype
##  exit
    end
```

Dynamically Built Param Statements

EQUEL/Fortran supports a special kind of dynamically built statement called a **param** statement. While the ability to supply names, expression values, and even entire qualifications in the form of host variables, as described in the *QUEL Reference Guide*, provides much dynamic flexibility, **param** statements considerably extend this flexibility. **Param** statements determine at runtime, not only the names, but also the number and data types of target-list elements. This feature, for example, allows construction of a completely general program that can operate on any table or form that you specify at runtime.

A general restriction on **param** statements is that you cannot use **param** target lists in **repeat** queries.

In EQUEL/Fortran, **param** versions are available for all statements in which:

- Assignments are made between host variables and database columns
- Assignments are made between host variables and form fields (or tablefield columns)

Not only **retrieve**, **append**, and **replace**, but also many forms-related statements such as **getform**, **putform**, **initialize**, **loadtable**, **insertrow**, and several others, have **param** versions.

Consider, again, the reason that these special versions of statements are needed. Non-**param** EQUEL statements, though relatively flexible in terms of substituting variables for expression constants, database and form object names, and entire **where** clauses, are nevertheless fixed at compile time in the number and data type of the objects to or from which assignment is made at runtime. Look at the following non-**param retrieve** statement, for example:

```
##   character*100  chvar
##   integer        intvar
##   real           rvar

##   character*25 table
##   character*25 col1, col2, col3

C  Assignments are made at runtime to all variables
C  declared in the two lines immediately above,
C  representing names of database objects. Then the
C  following RETRIEVE statement gets
C  data from the specified table and columns

##   retrieve (chvar = table.col1, intvar = table.col2,
##             rvar = table.col3)
```

In this example, the host variables, the table name and the names of all three columns represent all components of the target list. What cannot vary in this way of coding, however, is the fact that the **retrieve** statement gets values from exactly three columns, and you must hard-code the data types of those three columns into the program. **Param** statements allow you to transcend those restrictions.

Syntax of Param Statements

These statements are called **param** statements because of the **param** function in place of its target list. The **param** function has the following syntax:

param *(target_string, var_address_array)*

Thus, for example, a **param retrieve** statement might look like this:

```
##  retrieve (param (target, varadr))
##  where qstr
```

The *target_string* is a formatted target list string that can be either a Fortran character variable or a Fortran character constant. Normally it is a variable, since the purpose of this feature is to allow statements to be built at runtime. The *var_address_array* is an array of locations to which values are assigned at runtime. The elements in this array then hold the addresses of variables of appropriate types to receive or supply data for the table columns or form fields with which the **param** statement interacts.

The *target_string* looks like a regular target list expression, except where a Fortran variable intended to receive or supply data in an assignment would normally appear. In place of these names, the *target_string* contains symbolic type indicators representing the variables. For each of these type indicators appearing in the target list, there must be an address recorded in the corresponding element of the *var_address_array*, beginning with *var_address_array(1)*.

UNIX

If your system does not include the **loc** built-in function to obtain addresses of integer and real variables, you can use the **IInum( )** function provided with EQUEL.

VMS

You can use the **%LOC** built-in function to obtain addresses of integer and real variables.

Windows

The "loc" intrinsic function (or the "%loc" built-in function) is used to access the address of variables.

To obtain the address of a character variable, you can use the **IIstr( )** function (UNIX), **IIdesc()** function (VMS), or **IIdesc()** function (Windows) provided with EQUEL. Examples of these functions appear at the end of this section.

At runtime, EQUEL processes the statement by associating the variable addresses with the type indicators embedded in the *target_string*. Addresses must have been previously placed in the cells of the array in a sequence corresponding to the sequence of type indicators in the *target_string*, such that the statement can find a list of the correct number of Fortran variables of the correct type.

The variable-type indicators can be any of the following:

i2 two-byte integer (**integer*2**)
i4 four-byte integer (**integer*4**)
f4 four-byte floating-point number (**real*4**)
f8 eight-byte floating-point number (**real*8** or **double precision**)
c[N] **character string**

In the list above, the length specifier N is optional. The default length is the size of the character variable.

In this context, the format indicator must always agree with the Fortran variable that supplies or receives the data. This format does not necessarily need to be the same as that of the column where the data is stored in the database. Store data to be retrieved from, or inserted into, table columns of type **date** in character arrays with a minimum length of 25 in your program. Retrieve items of type **money** into program variables of type **real*4** or **real*8**.

When you reference ordinary character-string data in a **param** target list, use the **c** type indicator with or without specifying the number of characters to be assigned. The optional length specification has the following effect, depending on the kind of statement in which the target list appears:

- In an *input* statement, such as **append** or **putform**, the length specification, N, attached to a **c** type indicator, limits to N the number of bytes actually assigned from the Fortran character variable to the database or form object. The default is to assign as many bytes from the Fortran variable as can be accommodated in the database or form object after trimming trailing blanks.
- In an *output* statement, such as **retrieve** or **getform**, the length specification limits to N the number of bytes of actual data assigned from the database or form object to the Fortran character variable. In the absence of the length specifier, EQUEL writes into the variable as much of the data as the variable can hold. The Fortran character variable is always padded with blanks, if the length of the data is shorter than that of the variable.

The following examples show a **param append** statement:

UNIX

```
  program param

##  DECLARE

C Declare variables to be used for supplying data to
C the database

##   character*27     chvar
##   integer*4        intvar
##   real*8           rvar

C Declare variables for the PARAM target list, the array C
of variable addresses, and the database table to be
C used.

##   character*100    tlist
##   integer*4        varadr(3)
##   character*25     tblnam

C Now assign values to variables in order to set up
C the PARAM statements. In a real application, this
```

```
C would be done during the process of interacting with
C the user, as well as by obtaining information from
C the system catalogs, or from the FRS, about the
C number and data type of table columns. In this
C example, the assignments are hard-coded.

     tblnam = 'employee'

C The following target list is for use with the APPEND
C statement. Note that the type indicators appear on the
C right-hand side of the assignments. Column names
C appear on the left-hand side.

     tlist = 'empname=%c, empnum=%i4, salary=%f8'

C The next three statements assign, to an array of
C integers, the addresses of variables which will
C supply data for the APPEND statement. Note the use of
C the EQUEL functions 'IInum and IIstr' to access the
C address of the variables.

     varadr(1) = IIstr (chvar)
     varadr(2) = IInum(intvar)
     varadr(3) = IInum(rvar)

C Next, values are assigned to the data variables
C themselves. Again, in an actual application this
C would likely be done by interacting with the user.

     chvar       = 'Jane Swygart'
     intar       = 332
     rvar        = 37500.00

##   ingres personnel
##   append to tblnam (PARAM (tlist, varadr))
##   exit
     end
```

VMS

```
program param_example

##   declare

C Declare variables to be used for supplying data to
C the database

##   character*27       ch_var
##   integer*4          int_var
##   real*8             real_var

C Declare variables for the PARAM target list, the
C array of variable addresses, and the database table
C to be used.

##   character*100      targlist
##   integer*4          varaddr(3)
##   character*25       tablename

C Now assign values to variables in order to set up
C the PARAM statements. In a real application, this
C would be done during the process of interacting with
C the user, as well as by obtaining information from
C the system catalogs, or from the FRS, about the
C number and data type of table columns. In this
C example, the assignments are hard-coded.
```

```
    tablename = 'employee'

C The following target list is for use with the APPEND
C statement. Note that the type indicators appear on
C the right-hand side of the assignments. Column names
C appear on the left-hand side.

    targlist = 'empname=%c, empnum=%i4, salary=%f8'

C The next three statements assign, to an array of
C integers, the addresses of variables that will supply
C data for the APPEND statement.

C Note the use of the EQUEL function 'IIdesc' to access
C the address of the character variable.

                varaddr(1) = IIdesc (ch_var)
                varaddr(2) = %loc(int_var)
                varaddr(3) = %loc(real_var)

            C Next, values are assigned to the data variables
            C themselves. Again, in an actual application this
            C would likely be done by interacting with the user.

                ch_var = 'Jane Swygart'
                int_var = 332
                real_var = 37500.00

##  ingres personnel
##  append to tablename (PARAM (targlist, varaddr))
##  exit
    end
```

Windows

```
program param_example

##   declare

C Declare variables to be used for supplying data to
C the database

##   character*27       ch_var
##   integer*4          int_var
##   real*8             real_var

C Declare variables for the PARAM target list, the
C array of variable addresses, and the database table
C to be used.

##   character*100      targlist
##   integer*4          varaddr(4)
##   character*25       tablename
C Data types
    integer*4 DATE, MONEY, CHAR, VARCHAR, INT, FLOAT, C, TEXT
    parameter (DATE =    3,
     1      MONEY       =       5,
     2      CHAR=       20,
     3      VARCHAR =   21,
     4      INT =       30,
     5      FLOAT       =       31,
     6      C   =       32,
     7      TEXT=       37 )

C Now assign values to variables in order to set up
```

```
C the PARAM statements. In a real application, this
C would be done during the process of interacting with
C the user, as well as by obtaining information from
C the system catalogs, or from the FRS, about the
C number and data type of table columns. In this
C example, the assignments are hard-coded.

    tablename = 'employee'

C The following target list is for use with the APPEND
C statement. Note that the type indicators appear on
C the right-hand side of the assignments. Column names
C appear on the left-hand side.

    targlist = 'empname=%c, empnum=%i4, salary=%f8'

C The next three statements assign, to an array of
C integers, the addresses of variables that will supply
C data for the APPEND statement.

C Note the use of the EQUEL function 'IIdesc' to access
C the address of the character variable.
C The type and the length of the Fortran character variable
C needs to be supplied
                  varaddr(1) = IIdesc (ch_var, CHAR, LEN(ch_var))
                  varaddr(2) = %loc(int_var)
                  varaddr(3) = %loc(real_var)
C Next, values are assigned to the data variables
C themselves. Again, in an actual application this
C would likely be done by interacting with the user.

                  ch_var = 'Jane Swygart'
                  int_var = 332
                  real_var = 37500.00

##  ingres personnel
##  append to tablename (PARAM (targlist, varaddr))
##  exit
    end
```

Practical Uses of Param Statements

Most applications do not need **param** statements, because programs are usually intended for specific purposes and are based on databases whose designs are known at the time the programs are coded. **Param** statements are crucial mainly for *generic* programs. An example of such a program is QBF, the Ingres user-interface program capable of operating on any database, and any table, form, or joindef specified by the user.

It is difficult to illustrate practical examples of **param** statements because in an actual application, you must code to determine the name, number and data type of the objects to be manipulated in a **param** statement target list, in addition to the coding required to obtain or operate on data values. For an extended, practical example, see UNIX, VMS, Windows—An Interactive Database Browser Using Param Statements in this chapter. The target string and address array are customarily built from information obtained from various sources: the user, the **formdata** and **tabledata** statements, and the Ingres system catalogs. In an EQUEL/FORMS program, a typical scenario prompts the user for the name of a form to operate on, and then uses the **formdata** and **tabledata** statements to get name and type information about the fields. Subsequently, the various **param** target lists and address arrays the program needs would be built using this information. The examples here illustrate only the syntax of the **param** statements themselves, as well as simplified mechanics of setting up their component parts.

The example above, with a **param append**, is typical for an *input* statement, where values are being supplied to the database or form from program variables. Other input statements include **replace**, **initialize**, **putform**, **loadtable**, **putrow**, and so forth.

Output statements are similar, except that the type indicators appear on the left-hand side of the assignment statements in the **param** target list. In these statements, program variables receive data from the database or the form. Output statements include **retrieve**, **getform**, **finalize**, **unloadtable**, **getrow**, and so forth. For the format of the **param** target lists for cursor statements, see Param Versions of Cursor Statements in this chapter.

Indicator Variables in Param Statements

You can code **param** statements to accommodate data assigned to or from nullable columns and form fields. The syntax is analogous to that previously described, with the exception that, in the target string, type indicators are needed in place of both the data variable and the indicator variable. Since indicator variables are always 2-byte integers, you can use the **i2** type indicator for this purpose. A sample target list of a **param retrieve** statement, including indicator variables, might look like this:

```
tlist = '%c:%i2=e.empname, %f8:%i2=e.salary'
```

The *var_address_array* corresponding to this target list needs four cells, initialized in the following order:

a character-variable address
an address of an **integer*2**
an address of a **real*8**
an address of another **integer*2**

When the **retrieve** statement executes, one or both of the short variables can contain the value -1 if null data were present in that row of the table.

Using the Sort Clause in Param Retrieves

Unlike the non-**param** version of the **retrieve** statement, the **param** version has no application-supplied names for result columns. The non-**param retrieve** uses the same names as for the host variables used to receive the data, but in a **param retrieve** these names are not present in the statement. Only the type indicators are seen by the EQUEL runtime system when the **param retrieve** is executed.

In order to meet the need for result column names in the statement, Ingres generates internal names. If you want to include a **sort** clause in a **param retrieve**, you must use the internally generated result column names as arguments to the **sort** clause. These names are "ret_var1", "ret_var2", and so forth, named sequentially for all the result columns represented by type indicators in the target list. (Ignore null indicators in determining this sequence.) For example, assume a target list as in the previous section:

```
tlist = '%c:%i2=e.empname,%f8:%i2=e.salary'
```

If you want to **retrieve** and **sort by** the result column representing salary, you must supply the internal name "ret_var2" to the **sort** clause:

```
##   retrieve (param(tlist,varadr))
##   sort by ret_var2:d
```

This sorts by the second result column, in descending order.

Param Versions of Cursor Statements

There are **param** versions for cursor versions of the **retrieve** and **replace** statements. In the case of the cursor retrieve, the **param** target list is used in the **retrieve cursor** statement, not in the **declare cursor** statement. The non-**param retrieve cursor** target list is simply a comma-separated list of Fortran variables corresponding to the result columns identified in the **declare cursor** statement. Therefore, the target string in the **param** version is a comma-separated list of type indicators, optionally with associated type indicators for the null indicator variables.

When you code the **declare cursor** statement for use with the **param** version of **retrieve cursor**, you should take advantage of the fact that the entire target list in **declare cursor** can be replaced by a host character variable. This, in effect, allows the whole retrieve statement in **declare cursor** to be determined at runtime. Then, the components of the param **retrieve cursor** can be built dynamically for the associated **declare cursor** statement.

The target string for a **retrieve cursor** statement might look something like the following:

```
tlist = '%c:%i2,%f8:%i2'
```

This target list is appropriate for a **retrieve cursor** where the associated **declare cursor** retrieved two nullable columns, one character string and one floating-point value.

The **replace cursor** statement also supports a **param** version. Its target list looks the same as in the non-cursor version of **replace**.

The following is a somewhat expanded example, showing both the **declare cursor**, **retrieve cursor**, and **replace cursor**:

UNIX

```
program cursor

##      declare

C Declare variables to be used for supplying data
C to the database

##   character*25     chvar
##   integer          intvar
##   real*8           rvar
##   integer*2        nulind

C Declare variables for the various target lists and
C the arrays of variable addresses

##   character*100   delist
##   character*100   rtlist
##   character*100   rplist
##   integer*4       rtvadr(10)
##   integer*4       rpvadr(5)
##   integer         nomore, ingerr
     character*10   newpay

     nomore = 0
     ingerr = 0

##   ingres 'personnel'

C Assign values of target lists for DECLARE CURSOR,
C RETRIEVE CURSOR, and REPLACE CURSOR. The second and
C third of these have PARAM clauses. The first doesn't
C need one, as it transfers no data. In the target
C list for RETRIEVE CURSOR, a null indicator
C is included for the floating-point value.

     delist = 'employee.empname, employee.age,
               employee.salary'
     rtlist = '%c, %i4, %f8:%i2'
     rplist = 'salary=%f8'

C Assign pointer values to the address array for
C the RETRIEVE CURSOR statement.

     rtvadr(1) = IIstr(chvar)
     rtvadr(2) = IInum(intvar)
```

```
     rtvadr(3) = IInum(rvar)
     rtvadr(4) = IInum(nulind)

##   declare cursor cursor1 For Retrieve (delist)

##          For Direct Update of (salary)

##   open cursor cursor1

10   continue

     if ((ingerr .eq. 0) .and. (nomore .eq. 0)) then
##   retrieve cursor cursor1 (param(rtlist, rtvadr))
##   inquire ingres (ingerr = ERRORNO, nomore = ENDQUERY)

C If an Ingres error occurred, or if no more rows
C found for the cursor, break loop.

     if ((ingerr .eq. 0) .and. (nomore .eq. 0)) then

C If salary for this record is null, print name and age,
C prompt the user to enter the salary, and replace the
C value in that row. If salary is not null, print name,
C age, and salary.

           if (nulind .eq. -1) then
                 print *, chvar, intvar
                 write (*,50)
50               format (' Enter Salary: ',$)
                 accept 51, rvar
51               format (f8.2)
                 if (rvar .gt. 0) then
                     rpvadr(1) = IInum(rvar)
##                   replace cursor cursor1 (param
                          (rplist, rpvadr))
                 endif
              else
                 print *, chvar, intvar, rvar
              endif

          else if (ingerr .eq. 1) then
                print *, 'Error occurred, exiting ...'

          else if (nomore .eq. 1) then
              print *, 'No more rows'
          endif
    goto 10

##   close cursor cursor1

##   exit
     end
```

VMS

```
program param_cursor

##      declare

C Declare variables to be used for supplying data
C to the database

##   character*25              ch_var
##   integer                int_var
##   real*8                 real_var
##   integer*2                 null_ind
```

```
C Declare variables for the various target lists and
C    the arrays of variable addresses

##   character*100 decl_cursor_list
##   character*100 ret_cursor_list
##   character*100 repl_cursor_list
##   integer*4              ret_varaddr(10)
##   integer*4              repl_varaddr(5)
##   integer              thatsall, ingerror
     character*10  newsalary

     thatsall = 0
     ingerror = 0

##   ingres 'personnel'

C Assign values of target lists for DECLARE CURSOR,
C RETRIEVE CURSOR, and REPLACE CURSOR. The second and
C third of these have PARAM clauses. The first doesn't
C need one, as it transfers no data. In the target
C list for RETRIEVE CURSOR, a null indicator is
C included for the floating-point value.

decl_cursor_list
='employee.empname,employee.age,employee.salary'
  ret_cursor_list = '%c, %i4, %f8:%i2'
  repl_cursor_list = 'salary=%f8'

C Assign pointer values to the address array for
C the RETRIEVE CURSOR statement.

     ret_varaddr(1) = IIdesc(ch_var)
     ret_varaddr(2) = %loc(int_var)
     ret_varaddr(3) = %loc(real_var)
     ret_varaddr(4) = %loc(null_ind)

##   declare cursor cursor1 for retrieve
##     (decl_cursor_list)
##   for direct update of (salary)

##   OPEN CURSOR cursor1

     do while ((ingerror .eq. 0) .and. (thatsall .eq. 0))
##   retrieve cursor cursor1 (Param(ret_cursor_list,
##   ret_varaddr))
##   inquire ingres (ingerror = ERRORNO,
##   thatsall = ENDQUERY)

C If an Ingres error occurred, or if no more rows
C found for the cursor, break loop.

     if ((ingerror .eq. 0) .and. (thatsall .eq. 0)) then

C If salary for this record is null, print name and
C age, prompt the user to enter the salary, and
C replace the value in that row. If salary is not
C null, print name, age, and salary.

     if (null_ind .eq. -1) then
         print *, ch_var, int_var
         write (*,50)
50       format (' Enter Salary: ',$)
         accept 51, real_var
51       format (f8.2)
         if (real_var .gt. 0) then
```

```
            repl_varaddr(1) = %loc(real_var)
##          REPLACE CURSOR cursor1 (PARAM
             (repl_cursor_list, repl_varaddr))
        endif
      else
         print *, ch_var, int_var, real_var
      endif

    else if (ingerror .eq. 1) then
          print *, 'Error occurred, exiting ...'

    else if (thatsall .eq. 1) then
          print *, 'No more rows'

    endif   end do
##    close cursor cursor1

##    exit
      end
```

Windows

```
        program param_cursor

##      declare

C Declare variables to be used for supplying data
C to the database

##   character*25              ch_var
##   integer               int_var
##   real*8               real_var
##   integer*2               null_ind

C Declare variables for the various target lists and
C    the arrays of variable addresses

##   character*100 decl_cursor_list
##   character*100 ret_cursor_list
##   character*100 repl_cursor_list
##   integer*4               ret_varaddr(10)
##   integer*4               repl_varaddr(5)
##   integer              thatsall, ingerror
      character*10  newsalary

         thatsall = 0
         ingerror = 0

##   ingres 'personnel'

C Assign values of target lists for DECLARE CURSOR,
C RETRIEVE CURSOR, and REPLACE CURSOR. The second and
C third of these have PARAM clauses. The first doesn''t
C need one, as it transfers no data. In the target
C list for RETRIEVE CURSOR, a null indicator is
C included for the floating-point value.

      decl_cursor_list ='employee.empname,employee.age,employee.salary'
      ret_cursor_list = '%c, %i4, %f8:%i2'
      repl_cursor_list = 'salary=%f8'

C Assign pointer values to the address array for
C the RETRIEVE CURSOR statement.

      ret_varaddr(1) = IIdesc(ch_var, 20, LEN(ch_var))
      ret_varaddr(2) = %loc(int_var)
```

```
      ret_varaddr(3) = %loc(real_var)
      ret_varaddr(4) = %loc(null_ind)

##   declare cursor cursor1 for retrieve
##     (decl_cursor_list)
##   for direct update of (salary)

##   OPEN CURSOR cursor1

      do while ((ingerror .eq. 0) .and. (thatsall .eq. 0))
##   retrieve cursor cursor1 (Param(ret_cursor_list,
##   ret_varaddr))
##   inquire_ingres (ingerror = ERRORNO,
##   thatsall = ENDQUERY)

C If an Ingres error occurred, or if no more rows
C found for the cursor, break loop.

      if ((ingerror .eq. 0) .and. (thatsall .eq. 0)) then

C If salary for this record is null, print name and
C age, prompt the user to enter the salary, and
C replace the value in that row. If salary is not
C null, print name, age, and salary.

      if (null_ind .eq. -1) then
         print *, ch_var, int_var
         write (*,50)
50       format (' Enter Salary: ',$)
         accept 51, real_var
51       format (f8.2)
         if (real_var .gt. 0) then
             repl_varaddr(1) = %loc(real_var)
##           REPLACE CURSOR cursor1 (PARAM
##             (repl_cursor_list, repl_varaddr))
        endif
      else
         print *, ch_var, int_var, real_var
      endif

      else if (ingerror .eq. 1) then
          print *, 'Error occurred, exiting ...'

      else if (thatsall .eq. 1) then
          print *, 'No more rows'

      endif
      end do
##    close cursor cursor1

##    exit
      end
```

Runtime Error Processing

This section describes a user-defined EQUEL error handler.

Programming for Error Message Output

By default, all Ingres and forms system errors are returned to the EQUEL program, and default error messages are printed on the standard output device. As discussed in the *QUEL Reference Guide* and the *Forms-based Application Development Tools User Guide*, you can also detect the occurrences of errors by means of the program using the **inquire_ingres** and **inquire_frs** statements. (Use **inquire_frs** for checking errors after forms statements. Use **inquire_ingres** for all other EQUEL statements.)

This section discusses an additional technique that enables your program not only to detect the occurrences of errors, but also to suppress the printing of default Ingres error messages if you choose. The **inquire** statements detect errors but do not suppress the default messages.

This alternate technique entails creating an error-handling function in your program, and passing its address to the Ingres runtime routines. This makes Ingres automatically invoke your error handler whenever either an Ingres or a forms-system error occurs. You must declare your program error handler as follows:

```
integer function funcname (errorno)
integer errorno
...
end
```

You must pass this function to the EQUEL routine **IIseterr( )** for runtime bookkeeping using the Fortran statements:

```
external funcname
integer funcname
IIseterr(funcname);
```

This forces all runtime Ingres errors through your function, passing the Ingres error number as an argument. If you choose to handle the error locally and suppress Ingres error message printing the function should return 0; otherwise the function should return the Ingres error number received.

Avoid issuing any EQUEL statements in a user-written error handler defined to **IIseterr**, except for informative messages, such as **message, prompt, sleep** and **clear screen,** and messages that close down an application, such as **endforms** and **exit.**

The example below demonstrates a typical use of an error function to warn users of access to protected tables. It also passes through all other errors for default treatment.

```
    program errhnd
##  declare

    external locerr
    integer locerr

    . . .
```

```
##  ingres dbname
    IIseterr( locerr )

    . . .

##  exit
    end

    integer function locerr(errno)
    parameter (TBLPRO = 5003)
    integer errno

    if (errno .eq. TBLPRO) then
    print *, 'You are not authorized for this/ operation.'
    locerr = 0
    else
        locerr = errno
    endif
    return
    end
```

A more practical example would be a handler to catch deadlock errors. For deadlock, a reasonable handling technique in most applications is to suppress the normal error message and simply restart the transaction.

The following EQUEL program executes a Multi-Query Transaction and handles Ingres errors, including restarting the transaction on deadlock.

This example uses a program-defined error handler, rather than the **inquire_ingres** statement, to detect Ingres errors. This technique allows the normal Ingres error message to be suppressed in the case of deadlock, and the transaction to restart automatically without the user's knowledge.

UNIX

```
##    program mqterr
##    declare
      parameter (NOERR = 0)
      external errprc, tdone
      integer errprc, ingerr
      logical tdone
      common /errors/ ingerr
##    ingres dbname
C Set up test data
##    create item (name=c10, number=i4)
      call IIseterr( errprc )
      ingerr = NOERR
C     The following do-while loop will iterate until the
C transaction completes or fails: it restarts the
C transaction on deadlock.
10    continue
      if ( .not. tdone() ) then
            goto 10
##    end
C The function 'tdone' contains the multi-query
C transaction. The transaction consists of an APPEND,
C a REPLACE and a DELETE of a single row.
##    logical function tdone
##    declare
      external dlock
      logical dlock
##    begin transaction
```

```
##    append To item (name='Barbara', number=38)
      if (dlock()) then
           tdone = .false.
           return
      endif
##    replace item (number=39) where item.name='Barbara'
      if (dlock()) then
           tdone = .false.
           return
      endif
##    delete item where item.number=38
      if (dlock()) then
           tdone = .false.
           return
      endif
##    end transaction
##    destroy item
##    exit
      tdone = .true.
##    end

C The following routine differentiates deadlock from
C other errors. If the Ingres error is deadlock,
C the DBMS will automatically
C ABORT an existing MQT. If the error is not deadlock,
C this routine aborts the transaction and the program.

##    logical function dlock
##    declare
      parameter (EDLOCK = 4700)
      parameter (NOERR = 0)
      integer ingerr
      common /errors/ ingerr

      if (ingerr .gt. 0) then
          if (ingerr .eq. EDLOCK) then
              ingerr = NOERR
              dlock = .true.
              return
      else
              print *, 'Aborting -- Error #', ingerr
##            abort
##            exit
              stop
          endif
      endif

      dlock = .false.
      return
##    end

C The following is a user-defined error-handling routine.
C Returns 0 if the Ingres error is deadlock to
C prevent the runtime system from printing an error
C message.

      integer function errprc (errno)
      parameter (EDLOCK = 4700)
      integer errno
      integer ingerr
      common /errors/ ingerr
      ingerr = errno
      if (errno .eq. EDLOCK) then
          errprc = 0
      else
          errprc = errno
```

```
      endif
      return
      end
```

VMS

```
##    program mqterr
##    declare
      parameter (ERR_NOERROR = 0)
      external errproc, transdone
      integer errproc, ingerr
      logical transdone
      common /errors/ ingerr

##    ingres dbname

C Set up test data

##    create item (name=c10, number=i4)
      call IIseterr( errproc )
      ingerr = ERR_NOERROR

C The following do-while loop will iterate until the
C transaction completes or fails: it restarts the
C transaction on deadlock.

      do while ( .not. transdone() )
      end do
##    end

C The function 'transdone' contains the multi-query
C transaction. The transaction consists of an APPEND,
C a REPLACE and a DELETE of a single row.

##    logical function transdone
##    declare

      external deadlock
      logical deadlock

##    begin transaction
##    append to item (NAME='Barbara', number=38)
      if (deadlock() .eq. .true.) then
          transdone = .false.
          return
      endif

##    append To (number=39) WHERE item.name='Barbara'
      if (deadlock() .eq. .true.) then
          transdone = .false.
          return
      endif

##    delete item Where item.number=38
      if (deadlock() .eq. .true.) then
          transdone = .false.
          return
      endif

##    end transaction
##    destory item
##    exit
      transdone = .true.
##    end

C The following routine differentiates deadlock from
```

```
C other errors. If the Ingres error is deadlock,
C the DBMS will automatically ABORT an existing MQT.
C If the error is not deadlock, this routine aborts
C the transaction and the program.

##    logical function deadlock
##    declare
      parameter (ERR_ DEADLOCK = 4700)
      parameter (ERR_NOERROR = 0)
      integer ingerr
      common /errors/ ingerr

      if (ingerr .gt. 0) then
          if (ingerr .eq. ERR_DEADLOCK) then
              ingerr = ERR_NOERROR
              deadlock = .true.
              return
          else
              print *, 'Aborting -- Error #', ingerr
##              abort
##              exit
              stop
          endif
      endif

      deadlock = .false.
      return
##    end

C The following is a user-defined error-handling routine.
C Returns 0 if the Ingres error is deadlock to
C prevent the runtime system from printing an error
C message.

    integer function errproc (errorno)
    parameter (ERR_DEADLOCK = 4700)
    integer errorno
    integer ingerr
    common /errors/ ingerr

    ingerr = errorno
    if (errorno .eq. ERR_DEADLOCK) then
        errproc = 0
    else
        errproc = errorno
    endif
    return
    end
```

Windows

```
##    program mqterr
##    declare
      parameter (ERR_NOERROR = 0)
      external errproc, transdone
      integer errproc, ingerr
      logical transdone
      common /errors/ ingerr

##    ingres dbname

C Set up test data

##    create item (name=c10, number=i4)
      call IIseterr( errproc )
      ingerr = ERR_NOERROR
```

```
C The following do-while loop will iterate until the
C transaction completes or fails: it restarts the
C transaction on deadlock.

      do while ( .not. transdone() )
      end do
##    end

C The function 'transdone' contains the multi-query
C transaction. The transaction consists of an APPEND,
C a REPLACE and a DELETE of a single row.

##    logical function transdone
##    declare

      external deadlock
      logical deadlock

##    begin transaction
##    append to item (NAME='Barbara', number=38)
      if (deadlock() .eq. .true.) then
          transdone = .false.
          return
      endif

##    append to item (number=39) WHERE item.name='Barbara'
      if (deadlock() .eq. .true.) then
          transdone = .false.
          return
      endif

##    delete item Where item.number=38
      if (deadlock() .eq. .true.) then
          transdone = .false.
          return
      endif

##    end transaction
##    destroy item
##    exit
      transdone = .true.
##    end

C The following routine differentiates deadlock from
C other errors. If the Ingres error is deadlock,
C the DBMS will automatically ABORT an existing MQT.
C If the error is not deadlock, this routine aborts
C the transaction and the program.

##    logical function deadlock
##    declare
      parameter (ERR_ DEADLOCK = 4700)
      parameter (ERR_NOERROR = 0)
      integer ingerr
      common /errors/ ingerr

      if (ingerr .gt. 0) then
          if (ingerr .eq. ERR_DEADLOCK) then
              ingerr = ERR_NOERROR
              deadlock = .true.
              return
          else
              print *, 'Aborting -- Error #', ingerr
##              abort
##              exit
```

```
            stop
        endif
    endif

    deadlock = .false.
    return
##  end

C The following is a user-defined error-handling routine.
C Returns 0 if the Ingres error is deadlock to
C prevent the runtime system from printing an error
C message.

    integer function errproc (errorno)
    parameter (ERR_DEADLOCK = 4700)
    integer errorno
    integer ingerr
    common /errors/ ingerr

    ingerr = errorno
    if (errorno .eq. ERR_DEADLOCK) then
      errproc = 0
    else
      errproc = errorno
    endif
    return
    end
```

Precompiling, Compiling, and Linking an EQUEL Program

This section describes the EQUEL preprocessor for Fortran and the steps required to precompile, compile, and link an EQUEL program.

Generating an Executable Program

The following sections describe command line operations that you can use to turn your EQUEL source code program into an executable program. These commands preprocess, compile, and link your program.

The EQUEL Preprocessor Command

Use the following command line to invoke the Fortran preprocessor:

eqf {*flags*} {*filename*}

where *flags* are as follows:

Flag	Description
-d	Adds debugging information to the runtime database error messages EQUEL generates. The source file name, line number, and the erroneous statement itself are printed with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If the **-f** flag is specified without a *filename*, the output is sent to standard output, one screen at a time. If the **-f** flag is omitted, output is given the basename of the input file, suffixed ".f" (UNIX) or ".for" (VMS and Windows).
-**i***N*	Sets the default size of integers to *N* bytes. *N* is 2 or 4. The default is 4. If 2 is used, the **-12** flag (UNIX), the **noi4** qualifier (VMS), or the /integer_size:16 flag (Windows) must be used with the Fortran compiler.
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename***.lis**, where *file*name is the name of the input file.
-lo	Like **-l**, but the generated Fortran code also appears in the listing file.
-n. *ext*	Specifies the extension used for filenames in **## include** and **##** include inline statements in the source code. If **-n** is omitted, **include** filenames in the source code must be given the extension ".qf".
-o. *ext*	Specifies the extension the preprocessor gives to both the translated include statements in the main program and the generated output files. If this flag is not provided, the default extension is ".f" If you use this flag in combination with the **-o** flag, then the preprocessor generates the specified extension for the translated **include** statements, but does not generate new output files for the **include** statements.
-o	Directs the preprocessor not to generate output files for include files.

Flag	Description
	This flag does not affect the translated **include** statements in the main program. The preprocessor generates a default extension for the translated include file statements unless you use the **-o.** *ext* flag.
-s	Reads input from standard input and generate Fortran code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Ctrl D** (UNIX) or **Ctrl Z** (VMS and Windows).
-w	Prints warning messages.
-**?**	Shows the available command line options for **eqf**.

The EQUEL/Fortran preprocessor assumes that input files are named with the extension ".qf". To override this default, specify the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated Fortran statements with the same name and the extension ".f" (UNIX) or ".for" (VMS and Windows).

If you enter the command without specifying any flags or a filename, a list of available flags for the command is displayed.

The following table presents a range of the options available with **eqf**.

Eqf Command Examples

Command	Comment
eqf file1	Preprocesses "file1.qf" to "file1.for"
eqf -l file2.xf	Preprocesses "file2.xf" to "file2.for" and creates listing "file2.lis"
eqf -s	Accept input from standard input and write generated code to standard output
eqf -ffile4.out file4	Preprocesses "file4.qf" to "file4.out"
eqf	Displays a list of available flags for this command.

The Fortran Compiler

The preprocessor generates Fortran code. The generated code is in tab format, in which each Fortran statement follows an initial tab. (For information on the EQUEL format acceptable as *input* to the preprocessor, see EQUEL Statement Syntax for Fortran in this chapter.)

UNIX

You must use the **f77** command to compile this code. You can use most of the **f77** command line options. If you use the **-i2** flag to interpret **integer** and **logical** declarations as 2-byte objects, you must have run the EQUEL/Fortran preprocessor with the **-i2** flag.

The following example preprocesses and compiles the file "test1". The EQUEL/Fortran preprocessor assumes the default file extension ".qf".

```
$ eqf test1
$ f77 test1l.f
```

VMS

You should use the VMS **fortran** command to compile this code. You can use most of the **fortran** command line options. If you use the **noi4** qualifier to interpret **integer** and **logical** declarations as 2-byte objects, you must have run the EQUEL/Fortran preprocessor with the **-i2** flag. You must not use the **g_floating** qualifier if floating-point values in the file are interacting with Ingres floating-point objects. Note, too, that many of the statements that the preprocessor generates are nonstandard extensions provided by VAX/VMS. You should not attempt to compile with the **nof77** qualifier, which requires compatibility with Fortran-66.

The following example preprocesses and compiles the file "test1". The EQUEL/Fortran preprocessor assumes the default file extension ".qf".

```
$ eqf test1
$ fortran/list testl
```

Windows

Use the Windows **df** command to compile this code. The following compile options are required for Windows:

/name:as_is	Treat uppercase and lowercase letters as different.
/iface:nomixed_str_len_arg	Requests that the hidden lengths be placed in sequential order at the *end* of the argument list.
/iface:cref	Names are not decorated, the caller cleans the call stack and var args are supported.

If you use the **/integer_size:16** qualifier to interpret **integer** and **logical** declarations as 2-byte objects, you must have run the Fortran preprocessor with the **-i2** flag.

The following example preprocesses and compiles the file "test1." The Embedded SQL preprocessor assumes the default extension:

```
esqlf test1
df /compile_only /name:as_is /iface:nomixed_str_len_arg /iface:cref test1
```

Note: Check the Readme file for any operating system specific information on compiling and linking EQUEL/Fortran programs.

Linking an EQUEL Program - UNIX

EQUEL programs require procedures from several Ingres libraries. The libraries required are listed below and must be included in your compile or link command *after* all user modules. The libraries must be specified in the order shown in the examples that follow.

Programs Without Embedded Forms

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
f77 -o dbentry dbentry.o \
$II_SYSTEM/ingres/lib/libingres.a \
-lm -lc
```

Note that both the math library and the C runtime library must be included.

Ingres shared libraries are available on some Unix platforms. To link with these shared libraries replace "libingres.a" in your link command with:

```
-L $II_SYSTEM/ingres/lib -linterp.1 -lframe.1 -lq.1 \
     -lcompat.1
```

To verify if your release supports shared libraries check for the existence of any of these four shared libraries in the $II_SYSTEM/ingres/lib directory. For example:

```
ls -l $II_SYSTEM/ingres/lib/libq.1.*
```

Programs with Embedded Forms

If your program includes embedded forms, you must link your program with some additional libraries. The following example demonstrates the link command of an EQUEL program called "formentry" that includes forms statements:

```
f77 -o formentry formentry.o \
$II_SYSTEM/ingres/lib/libingres.a  \
-lm -lc
```

Note that both the math library and the C runtime library must be included.

Compiling and Linking Precompiled Forms

The technique of declaring a precompiled form to the FRS is discussed in the *Embedded Forms Programming Guide*. To use such a form in your program, you must also follow the steps described here. In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in C. VIFRED lets you select the name for the file. Once you have created the C file this way, you can compile it into linkable object code with the **cc** command:

cc *filename*

The output of this command is a file with the extension ".o". You then link this object file with your program by listing it in the link command, as in the following example, which includes the compiled form "empform.o":

```
f77 -o formentry formentry.o \
empform.o \
 $II_SYSTEM/ingres/lib/libingres.a \
  -lm -lc
```

Linking an EQUEL Program - VMS

EQUEL programs require procedures from several VMS shared libraries in order to run properly. After preprocessing and compiling an EQUEL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
$ link dbentry.obj,-
 ii_system:[ingres.files]equel.opt/opt
```

It is recommended that you do not explicitly link in the libraries referenced in the EQUEL.OPT file. The members of these libraries change with different releases of Ingres. Consequently, you can be required to change your link command files in order to link your EQUEL programs.

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *Embedded Forms Programming Guide.* To use such a form in your program, you must also follow the steps described here. In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry" by listing it in the link command, as in the following example:

```
$ link formentry,-
 empform.obj,-
 ii_system:[ingres.files]equel.opt/opt
```

Linking an EQUEL Program without Shared Libraries

While the use of shared libraries in linking EQUEL programs is recommended for optimal performance and ease-of-maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by EQUEL are listed in the equel.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
$ link dbentry,-
 ii_system:[ingres.files]equel.noshare/opt
```

Linking an EQUEL Program - Windows

To run properly, EQUEL programs require procedures from several Windows libraries. After preprocessing and compiling an EQUEL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
link /out:dbentry.exe, \
 dbentry.obj,\
 %II_SYSTEM%\ingres\lib\libingres.lib
```

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *Forms-based Application Development Tools User Guide*. To use such a form in your program, you must also follow the steps described here. Within VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the C language. VIFRED lets you select the name for the file. Once you have created the C file this way, you can compile it into linkable object code with the command

cl –c –MD *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry" by listing it in the link command, as in the following example:

```
link /out:formentry.exe, \
 empform.obj,\
 %II_SYSTEM%\ingres\lib\libingres.lib
```

Include File Processing

The EQUEL **include** statement provides a means to include external files in your program's source code. Its syntax is:

include *filename*

Filename is a quoted string constant specifying a file name, a system environment variable (UNIX and Windows) or logical name (VMS) that points to the file name. If no extension is given to the filename (or to the file name pointed at by the environment variable (UNIX and Windows) or defined as logical (VMS), the default Fortran input file extension ".qf" is assumed.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *QUEL Reference Guide.*

UNIX

The included file is preprocessed and an output file with the same name, but with the default output extension ".fish generated. You can override this default output extension with the **-o**.*ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified **include** output file. If you use the **-o** flag with no extension, no output file is generated for the include file.

If you use both the **-o.***ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program. However, it does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## include "employee.qf"
```

is preprocessed to the Fortran statement:

```
include 'employee.f'
```

and the file "employee.qf" is translated into the Fortran file "employee.f."

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
## INCLUDE "MYDECLS";
```

The name "MYDECLS" can be defined as a system environment variable pointing to the file "/dev/headers/myvars.qf" by means of the following command at the system level:

```
setenv MYDECLS "/dev/headers/myvars"
```

Assume now that "inputfile" is preprocessed with the command:

```
esqlf -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Fortran statement:

```
include '/dev/headers/myvars.h'
```

and the Fortran file "/dev/headers/myvars.h" is generated as output for the original include file, "/dev/headers/myvars.qf".

You can also specify include files with a relative path. For example, if you preprocess the file "/dev/mysource/myfile.qf," the EQUEL statement:

```
## include "../headers/myvars.qf"
```

is preprocessed to the Fortran statement:

```
include '../headers/myvars.f'
```

and the Fortran file "/dev/headers/myvars.f" is generated as output for the original include file, "/dev/headers/myvars.qf". This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *QUEL Reference Guide.*

VMS

If you specify a different extension with the **-n** flag of the **eqf** statement, then you must also specify *filename* with that extension.

The included file is preprocessed and an output file with the same name but with the default output extension ".for" is generated. You can override this default output extension with the **-o**.*ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified **include** output file. If you use the **-o** flag without an extension, no output file is generated for the include file. This is useful for program libraries that are using VMS MMS or MAKE dependencies.

If you use both the **-o**.*ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program but does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## include "employee.qf"
```

is preprocessed to the Fortran statement:

```
include 'employee.for'
```

and the employee.qf file is translated into the Fortran file "employee.for."

As another example, assume that a source file called "inputfile" contains the **include** statement shown below.

```
## include "mydecls"
```

The name "mydecls" is defined as a system logical name pointing to the file "dra1:[headers]myvars.qf" by means of the following command at the DCL level:

```
$ define mydecls dra1:[headers]myvars.qf
```

Assume now that "inputfile" is preprocessed with the command:

```
$ eqf -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Fortran statement:

```
include 'dra1:[headers]myvars.h'
```

and the Fortran file "dra1:[headers]myvars.h" is generated as output for the original include file, "dra1:[headers]myvars.qf."

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.qf," the EQUEL statement:

```
## include "[-.headers]myvars.qf"
```

is preprocessed to the Fortran statement:

```
# include '[-.headers]myvars.for'
```

and the Fortran file "dra1:[headers]myvars.for" is generated as output for the original include file, "dra1:[headers]myvars.qf."

Windows

If you specify a different extension with the **-n** flag of the **eqf** statement, then you must also specify *filename* with that extension.

The included file is preprocessed and an output file with the same name but with the default output extension ".for" is generated. You can override this default output extension with the **-o**.*ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified **include** output file. If you use the **-o** flag without an extension, no output file is generated for the include file.

If you use both the **-o**.*ext* and the **-o** flags, then the preprocessor generates the specified extension for the translated **include** statements in the program but does not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## include "employee.qf"
```

is preprocessed to the Fortran statement:

```
include 'employee.for'
```

and the employee.qf file is translated into the Fortran file "employee.for."

As another example, assume that a source file called "inputfile" contains the **include** statement shown below.

```
## include "mydecls"
```

The name "mydecls" is defined as a system environment name pointing to the file "c:\headers\myvars.qf" by means of the following command at the command prompt:

```
set mydecls=c:\headers\myvars.qf
```

Assume now that "inputfile" is preprocessed with the command:

```
$ eqf -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Fortran statement:

```
include 'c:\headers\myvars.h'
```

and the Fortran file "c:\headers\myvars.h" is generated as output for the original include file, "c:\headers\myvars.qf."

You can also specify include files with a relative path. For example, if you preprocess the file "c:\mysource\myfile.qf," the EQUEL statement:

```
## include "..\headers\myvars.qf"
```

is preprocessed to the Fortran statement:

```
# include '..\headers\myvars.for'
```

and the Fortran file "c:\headers\myvars.for" is generated as output for the original include file, "c:\headers\myvars.qf." 

Including Source Code with Labels

Some EQUEL statements generate labels (statement numbers). The statement numbers 7000 through 12000 are reserved for the preprocessor. If you include a file containing statements that generate labels, you must be careful to include the file only once in a given Fortran scope. Otherwise, you may find that the compiler later complains that the generated labels are defined more than once in that scope.

The statements that generate labels are the **retrieve** statement and all the EQUEL/FORMS block-type statements, such as **display** and **unloadtable**.

Coding Requirements for Writing EQUEL Programs

The following sections describe coding requirements for writing EQUEL programs.

Comments Embedded in Fortran Output

Each EQUEL statement generates one comment and a few lines of Fortran code. You may find that the preprocessor translates 50 lines of EQUEL into 200 lines of Fortran. This may result in confusion about line numbers when you debug the original source code. To facilitate debugging, each group of Fortran statements associated with a particular statement is preceded by a comment corresponding to the original EQUEL source. (Only *executable* EQUEL statements are preceded by a comment.) Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

EQUEL Statements and Fortran If Blocks

Because each EQUEL statement may generate several Fortran statements, you must use the block-style Fortran **if** statement to conditionally transfer control to EQUEL statements. For example:

```
if (error .eq. .true.) then
## message "Error on update"
## sleep 2
 end if
```

EQUEL also generates many nested constructs of **do** loops and **if** blocks specifically for block-structured statements, such as **display** and **unloadtable**. If you omit an **end if** from your Fortran source, the Fortran compiler complains that there is a missing **end** statement, which you may trace back to a preprocessor-generated **if**.

You can solve this problem by checking for matching **if-end** pairs in the original EQUEL/Fortran source file.

EQUEL Statements that Generate Labels

As mentioned, some EQUEL statements generate labels. These are the **retrieve** statement and all the EQUEL/FORMS block-type statements. Each of these statements reserves its own range of 200 labels in an overall range for such statements of 7000 through 12000. Consequently, you cannot have more than 200 of any single label-generating statement in the same program unit. For example, 201 **display** statements in a single subroutine will cause a compiler error indicating that a particular label has been used more than once. You could, however, have 200 **display** statements and 200 **unloadtable** statements without causing a problem.

An EQUEL Statement that Does Not Generate Code

The **declare cursor** statement does not generate any Fortran code. This statement should not be coded as the only statement in Fortran constructs that do not allow *null* statements.

EQUEL/Fortran Preprocessor Errors

To correct most errors, you may wish to run the EQUEL preprocessor with the listing (**-l**) option on. The listing helps locate the source and reason for the error.

For preprocessor error messages specific to the Fortran language, see the next section.

Preprocessor Error Messages

The following is a list of error messages specific to the Fortran language.

E_E10001 Unsupported Fortran type '%0c' used. **Double** assumed. Ingres does not support the Fortran types **complex** and **double** complex

Explanation: There is no Ingres type corresponding to **complex** or **double complex**, so the preprocessor does not map this declaration to an Ingres type. The preprocessor will continue to generate code as if you had declared the variable in question to be of type **double precision**.

If you want to store the two **real** (or **double precision**) components of a **complex** (or **double complex**) variable then declare a pair of real (or **double precision**) variables to the preprocessor, copy the components to them, and then store the copies.

E_E10002 Fortran **parameter** can only be used with values. Type names, variable names, and **parameter** names are not allowed.

Explanation: You have used the Fortran "**parameter** name = value" statement, but "value" is not an integer constant, a floating constant, or a string constant. You may have used the name of a Fortran data type, or a variable (or **parameter**) name instead of one of the legal constant types. If you do wish Ingres to know about this name then you must change the "value" to be a constant.

E_E10003 Incorrect indirection on variable '%0c'. The variable is declared as an array and is not subscripted, or is subscripted but is not declared as an array (%1c,%2c).

Explanation: This error occurs when the value of a variable is incorrectly expressed because of faulty indirection. For example, the name of an integer array has been given instead of a single array element, or, in the case of string variables, a single element of the string (that is, a character) has been given instead of the name of the array. The preprocessor will continue to generate code, but the program will not execute correctly if it is compiled and run.

E_E10004 Last Fortran structure field referenced in '%0c' is unknown.

Explanation: This error occurs when the preprocessor encounters an unrecognized name in a structure reference. The preprocessor will continue to generate code, but this statement will either cause a runtime error or produce the wrong result if the resulting program is compiled and run.Check for misspellings in field names and ensure that all of the structure fields have been declared to the preprocessor.

E_E10005 Unclosed Fortran block - %0c unbalanced **end**(s).

Explanation: The preprocessor reached the end of the file still expecting one or more closing **end** statements. Make sure that you have no '**end**s' in an unclosed character or string constant, or have not accidentally commented out a closing **end**. Balance each **subroutine**, or **function** statement with a closing **end**.

E_E10006 Unsupported definition of nested Fortran function '%0c'.

Explanation: EQUEL read the beginning of a subprogram (**program**, subroutine, or **function**) while still in a previous subprogram definition.

Ensure that the **end** statement for a previous subprogram definition is not missing.

E_E10007 No ## **declare** before first EQUEL/Fortran statement '%0c'.

Explanation: You must issue a ## **declare** statement before any Ingres statement. The generated code will probably not compile.

E_E10008 Reissue of ## **declare** in Fortran program unit. The second declaration is ignored.

Explanation: The **declare** statement should be issued exactly once in each Fortran program unit. This error can also be caused by forgetting to ## the **program**, **subroutine**, or **function** line (and the matching **end**). EQUEL will ignore the extraneous **declare** statement.

E_E10009 No ## **declare forms** before forms statement '%0c' in Fortran program unit.

Explanation: You must issue a ## **declare forms** statement before any **forms** statement. The generated will probably not compile.

E_E1000A Undefined structure name '%0c' used in **record** declaration.

Explanation: You have declared a **record** variable using the name of a structure that is unknown to the preprocessor.
The preprocessor will continue to generate code, but the resulting program will not run properly. If you do not use this variable with an Ingres statement then remove the record declaration. Otherwise ensure that the corresponding structure declaration is made known to the preprocessor

E_E1000C Illegal length specified for Fortran numeric variable.

Explanation: Fortran integer variables may be 1, 2, or 4 bytes, and floating-point variables may be either 4 or 8 bytes. Specifying any other value is illegal.

Sample Applications

This section contains several sample applications for the UNIX, VMS, and Windows environments.

UNIX and VMS—The Department-Employee Master/Detail Application

This application runs in a master/detail fashion, using two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table to reduce expenses. Department information is stored in program variables. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo (the Toberesolved database table, described below) to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second is for the Employee table. The **create** statements used to create the tables are shown below. The cursors retrieve all the information in their respective tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, both from the Department table and the Employee table, is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the EQUEL statements. The program illustrates table creation, multi-query transactions, all cursor statements and direct updates. For purposes of brevity, error handling on data manipulation statements is simply to close down the application.

The application runs in the UNIX and VMS environments.

UNIX

The following two **create** statements describe the Employee and Department database tables:

```
##    create dept
##        (name        = c1      /* Department name */
##         totsales    = money,  /* Total sales */
##         employees   = i2)     /* Number of employees */

##    create employee
##        (name        = c20,    /* Employee name */
##         age         = i1,     /* Employee age */
```

```
##         idno        = i4,     /* Unique employee id */
##         hired       = date,   /* Date of hire */
##         dept        = c10,    /* Employee department */
##         salary      = money)  /* Yearly salary */

C
C   Procedure: MAIN
    Purpose:   Main body of the application. Initialize the
C              database, process each department,
C              and terminate the session.
C

       program main

       print *, 'Entering application to process expenses.'
       call InitDb()
       call PrcDpt()
       call EndDb()
       print *, 'Successful completion of application.'
       end

C
C   Procedure: InitDb
C   Purpose:   Initialize the database.
C              Start up the database, and abort if an error.
C              Before processing employees, create the table for
C              employees who lose their department, 'toberesolved'.
C              Initiate the multi-statement transaction.
C

##     subroutine InitDb()

##     declare

##     integer*4 errnum
##     character*256 errtxt

       external ErrEnd
       integer*4 ErrEnd

##     ingres personnel

       print *, ' Creating "To_Be_Resolved" table.'
##     create toberesolved
##          (name   = char(20),
##           age    = smallint,
##           idno   = integer,
##           hired  = date,
##           dept   = char(10),
##           salary = money)

##      inquire_ingres (errnum = errorno)
        if (errnum .NE. 0) then
##          inquire_ingres (errtxt = ERRORTEXT)
            print *, ' Fatal error on creation:'
            print *, errtxt
##          exit
            call exit(-1)
        endif

C
C    Inform Ingres runtime system about error handler.
C    All subsequent errors close down the application.
C
           call IIserr(ErrEnd)
```

```
##         begin transaction

##         end

C
C   Procedure: EndDb
C   Purpose:   Close off the multi-statement transaction and access
C              to the database after successful completion of the
C              application.
C
##         subroutine EndDb()
##         declare

##         end transaction
##         exit
##         end

C
C   Procedure: PrcDpt
C   Purpose:   Scan through all the departments, processing each
C              one. If the department has made less than
C              $50,000 in sales, then the department is dissolved.
C              For each department process all the employees
C              (they may even be moved to another table).
C              If an employee was terminated,
C              then update the department'employee counter. No
C              error checking is done for cursor updates.
C

##         subroutine PrcDpt()

##         declare

C    Corresponds to the 'dept' table
##         character*12      dptnam
##         double precision  dptsal
##         integer*2         dptemp

C                  Cursor loop control
##         integer*4         nmrows
C                  Minimum sales goal for department
##         parameter         (SALMIN = 50000.00)
C                  Number of terminated employees
##         integer*2         emptrm
C                  Department deleted indicator
           integer*2         deldpt
C                  Formatting value
           character*21      dptfmt

           emptrm = 0
           nmrows = 0

##     range of d is dept
##     declare cursor deptcsr for
##        retrieve (d.name, d.totsales, d.employees)
##        for direct update of (name, employees)

##     open cursor deptcsr

100    continue

##          retrieve cursor deptcsr
##              (dptnam, dptsal, dptemp)
##          inquire_equel (nmrows = ENDQUERY)

           if (nmrows .EQ. 0) then
```

```
C      Did the department reach minimum sales?
                if (dptsal .LT. SALMIN) then

##                  delete cursor deptcsr

                    deldpt = 1
                    dptfmt = ' -- DISSOLVED --'
                else
                    deldpt = 0
                    dptfmt = ' '
                endif

C      Log what we have just done

            print 11, dptnam, dptsal, dptfmt 11
            format (' Department: ', a14, ', Total Sales: ',
                            f12.3, a)

C      Now process each employee in the department
            call PrcEmp( dptnam, deldpt, emptrm )

C      If some employees were terminated, record this fact
            if (emptrm .GT. 0 .AND. deldpt .EQ. 0) then
##              replace cursor deptcsr
##                   (employees = dptemp - emptrm)
            endif

            endif
        if  (nmrows .EQ. 0) goto 100

##      close cursor deptcsr
##      end

C
C   Procedure: PrcEmp
C   Purpose:   Scan through all the employees for a particular
C              department. Based on given conditions the employee
C              may be terminated or    given a salary reduction.
C              1.    If an employee was hired since 1985 then the
C                    employee is terminated.
C              2.    If the employee's yearly salary is more than
C                    the minimum   company wage of $14,000 and the
C                    employee is not close to Retirement (over 58
C                    years of age), then the employee takes a 5%
C                    salary reduction.
C              3.    If the employee's department is dissolved and
C                    the employee is not terminated, then the
C                    employee is moved into the
C                    'toberesolved' table.
C   Parameters:
C              dptnam     -  Name of current department.
C              deldpt     -  Is current department being dissolved?
C              emptrm     -  Set locally to record how many
C                            employees were terminated for the
C                            current department.
C

##      subroutine PrcEmp( dptnam, deldpt, emptrm )
##      character*12   dptnam
        integer*2      deldpt
C                 Number of terminated employees
        integer*2      emptrm

##      declare
```

```
C    Corresponds to 'employee' table
##     character*20   empnam
##     integer*2      empage
##     integer*4      empid
##     character*25   emphir
##     real*4         emppay
##     integer*4      emp85

C             Cursor loop control
##     integer*4      nmrows
C             Minimum employee salary
##     parameter      (MINPAY = 14000.00)
C             Age of employees near to retirement
##     parameter      (NEAR65 = 58)
C             Percentage of current salary to receive
##     parameter      (SALRED = 0.95)
C             Formatting values
       character*14   title
       character*25   desc

       nmrows = 0
C
C    Note the use of the Ingres function to find out who was
C    hired since 1985.
C
##        range of e is employee
##        declare cursor empcsr for
##        retrieve (e.name, e.age, e.idno, e.hired, e.salary, res =
##        int4(interval('days', e.hired - date('01-jan-1985'))))
##               where e.dept = dptnam
##               for direct update of (name, salary)
##        open cursor empcsr

       emptrm = 0
10     continue

##     retrieve cursor empcsr (empnam, empage, empid,
##           emphir, emppay, emp85)
##     inquire_equel (nmrows = ENDQUERY)

       if (nmrows .EQ. 0) then

          if (emp85 .GT. 0) then

##            delete cursor empcsr

              title = 'Terminated:'
              desc = 'Reason: Hired since 1985.'
              emptrm = emptrm + 1

          else

C      Reduce salary if not near retirement
          if (emppay .GT. MINPAY) then
                if (empage .LT. NEAR65) the

##                     replace cursor empcsr
##                       (salary = salary * SALRED)

                    title = 'Reduction: '
                    desc = 'Reason: Salary.'

C      Do not reduce salary
                else
                    title = 'No Changes:'
                    desc = 'Reason: Retiring.'
```

```
                endif

C      Make no changes in salary
             else
                title = 'No Changes:'
                desc = 'Reason: Salary.'
             endif

C      Was employee's department dissolved ?
             if (deldpt .NE. 0) then
##               append to toberesolved (e.all)
##                    where e.idno = empid

##               delete cursor empcsr
             endif
          endif

C      Log the employee's information
          print 12, title, empid, empnam, empage, emppay,
      &               desc
12          format (' ', a, ' ', i6, ', ', a, ', ', i2, ', ', f8.2,
      &                   ';', ' ' a)
       endif

       if (nmrows .EQ. 0) goto 10

##       close cursor empcsr

##     end
C
C   Procedure: ErrEnd
C   Purpose:   If an error occurs during the execution of an EQUEL
C           statement, this error handler is called. Errors are
C           printed and the current database session is terminated.
C           Any open transactions are implicitly closed.
C   Parameters:
C           ingerr - Integer containing Ingres error number.
C
##    integer function ErrEnd(ingerr)
      integer*4 ingerr

##    declare
##    character*256 errtxt

##    inquire_ingres (errtxt = errortext)
      print *, ' Closing down because of database error:'
      print *, errtxt

##    abort
##    exit
      call exit(-1)

      ErrEnd = 0
##    end
```

VMS

The following two **create** statements describe the Employee and Department database tables:

```
##          create dept
##             (name       = c12,    /* Department name */
##              totsales   = money,  /* Total sales */
##              employees  = i2)     /* Number of employees */

##    create employee
```

```
##              (name       = c20,    /* Employee name */
##                 age        = i1,     /* Employee age */
##                 idno       = i4,     /* Unique employee id */
##                 hired      = date,   /* Date of hire */
##                 dept       = c10,    /* Employee department */
##                 salary     = money)  /* Yearly salary */

!
!  Procedure:  MAIN
!  Purpose:    Main body of the application. Initialize the
!              database, process each department, and terminate
!              the session.
!

        program main

        print *, 'Entering application to process expenses.'
        call Init_Db()
        call Process_Depts()
        call End_Db()
        print *, 'Successful completion of application.'
        end

!
!  Procedure:  Init_Db
!  Purpose:    Initialize the database.
!              Start up the database, and abort if an error.
!              Before processing employees, create the table for
!              employees who lose their department,
!              'toberesolved'. Initiate the multi-statement
!              transaction.
!
##      subroutine Init_Db()

##      declare

##      integer*4 err_no
##      character*256 err_text

        external Close_Down
        integer*4 Close_Down

##      ingres personnel

        print *, ' Creating "To_Be_Resolved" table.'
##      create toberesolved
##          (name   = c20,
##           age    = smallint,
##           idno   = integer,
##           hired  = date,
##           dept   = c10,
##           salary = money)

##      inquire_ingres (err_no = errorno)
        if (err_no .NE. 0) then
##            inquire_ingres (err_text = errortext)
              print *, ' Fatal error on creation:'
              print *, err_text
##            exit
              call exit(-1)
        endif

            !
            ! Inform Ingres runtime system about error handler.
            ! All subsequent errors close down the application.
            !
```

```
      call IIseterr(Close_Down)

##    begin transaction

##    end

!
!  Procedure:  End_Db
!  Purpose:    Close off the multi-statement transaction and
!              access to the database after successful completion
!              of the application.
!

##      subroutine End_Db()
##      declare

##      end transaction
##      exit
##      end

!
!  Procedure: Process_Depts
!  Purpose:   Scan through all the departments, processing each
!             one. If the department has made less than $50,000
!             in sales, then the department is dissolved. For
!             each department process all the employees (they may
!             even be moved to another table).If an employee was
!             terminated, then update the department's employee
!             counter. No error checking is done for cursor
!             updates.
!

##    subroutine Process_Depts()

##    declare

##    structure /department/ !Corresponds to the 'dept' table
##            character*12 name
##            double precision totsales
##            integer*2 employees
##    end structure
##    record /department/ dpt

##    integer*4 no_rows                      ! Cursor loop control
##    parameter MIN_DEPT_SALES = 50000.00  ! Min department sales
##    integer*2 emps_term                    ! Employees terminated
      integer*2 deleted_dept               ! Was the dept deleted?
      character*21 dept_format             ! Formatting value

      emps_term = 0
      no_rows = 0

##    range of d is dept
##    declare cursor deptcsr for
##              retrieve (d.name, d.totsales, d.employees)
##              for direct update of (name, employees)
##    open cursor deptcsr

      do while (no_rows .EQ. 0)

##        retrieve cursor deptcsr
##              (dpt.name, dpt.totsales, dpt.employees)
##        inquire_equel (no_rows = endquery)

          if (no_rows .EQ. 0) then
```

```
                ! Did the department reach minimum sales?
                if (dpt.totsales .LT. MIN_DEPT_SALES) then

##                  delete cursor deptcsr

                    deleted_dept = 1
                    dept_format = ' -- DISSOLVED --'
                else
                    deleted_dept = 0
                    dept_format = ' '
                endif

                ! Log what we have just done
                ! Log what we have just done
                print 11, dpt.name, dpt.totsales, dept_format
11              format
              (' Department: ', a14, ', Total Sales: ', f12.3, a)

                ! Now process each employee in the department
                call Process_Employees( dpt.name, deleted_dept,
     1                  emps_term )

             ! If some employees were terminated, record this fact
                if (emps_term .GT. 0 .AND.
     1              deleted_dept .EQ. .FALSE.) then
##                  replace cursor deptcsr
##                      (employees = dpt.employees - emps_term)
                endif

            endif
       end do

##     close cursor deptcsr
##     end

!
!  Procedure: Process_Employees
!  Purpose:   Scan through all the employees for a particular
!             department. Based on given conditions the employee
!             may be terminated, or given a salary reduction.
!             1. If an employee was hired since 1985 then the
!                employee is terminated.
!             2. If the employee's yearly salary is more than
!                the minimum company wage of $14,000 and the
!                employee is not close to retirement (over 58
!                years of age), then the employee takes a 5%
!                salary reduction.
!             3. If the employee's department is dissolved and
!                the employee is not terminated, then the
!                employee is moved into the 'toberesolved' table.
! Parameters:
!             dept_name     - Name of current department.
!             deleted_dept  - Is current department being
!                             dissolved?
!             emps_term     - Set locally to record how many
!                             employees were terminated for
!                             the current department.
!
!

##        subroutine Process_Employees( dept_name, deleted_dept,
     1                       emps_term )
##        character*12 dept_name
          integer*2 deleted_dept
          integer*2 emps_term
```

```
##        declare

##        structure /employee/ !Corresponds to 'employee' table
##                  character*20  name
##                  integer*2     age
##                  integer*4     idno
##                  character*25  hired
##                  real*4        salary
##                  integer*4     hired_since_85
##        end structure
##        record /employee/ emp

##        integer*4 no_rows                  ! Cursor loop control
##        parameter MIN_EMP_SALARY = 14000.00
!                                           Minimum employee salary
##        parameter NEARLY_RETIRED = 58
##        parameter SALARY_REDUC   = 0.95
          character*14 title                  ! Formatting values
          character*25 description

          no_rows = 0
          !
          ! Note the use of the Ingres function to find
          ! out who was hired since 1985.
          !
##        range of e is employee
##        declare cursor empcsr for
##           retrieve (e.name, e.age, e.idno, e.hired, e.salary,
##              res = int4(interval('days', e.hired -
##              date('01-jan-1985'))))
##              where e.dept = dept_name
##              for direct update of (name, salary)

##        open cursor empcsr

          emps_term = 0      ! Record how many
          do while (no_rows .EQ. 0)

##            retrieve cursor empcsr (emp.name, emp.age, emp.idno,
##                  emp.hired, emp.salary, emp.hired_since_85)
##            inquire_equel (no_rows = endquery)

              if (no_rows .EQ. 0) then

                   if (emp.hired_since_85 .GT. 0) then

##                    delete cursor empcsr

                      title = 'Terminated:'
                      description = 'Reason: Hired since 1985.'
                      emps_term = emps_term + 1

                   else
                      ! Reduce salary if not nearly retired
                      if (emp.salary .GT. MIN_EMP_SALARY) then

                          if (emp.age .LT. NEARLY_RETIRED) then

##                              replace cursor empcsr
##                                (salary = salary * SALARY_REDUC)

                                title = 'Reduction: '
                                description = 'Reason: Salary.'

                          else
```

```
                                ! Do not reduce salary
                                title = 'No Changes:'
                                description = 'Reason: Retiring.'
                            endif

                    else ! Leave employee alone

                            title = 'No Changes:'
                            description = 'Reason: Salary.'
                    endif

                    ! Was employee's department dissolved ?
                    if (deleted_dept .NE. 0) then
##                          append to toberesolved (e.all)
##                               where e.idno = emp.idno

##                          delete cursor empcsr
                    endif
                endif

                ! Log the employee's information
                print 12, title, emp.idno, emp.name, emp.age,
                            emp.salary,
     1              description
12              format (' ', a, ' ', i6, ', ', a, ', ', i2, ', ',
     1              f8.2, ';', 1 ' ' a)

            endif
        end do

##      close cursor empcsr

##      end

!
!  Procedure:  Close_Down
!  Purpose:    If an error occurs during the execution of an
!              EQUEL statement, this error handler is called.
!              Errors are printed and the current database session
!              is terminated.
!              Any open transactions are implicitly closed.
!  Parameters:
!              ingerr - Integer containing Ingres error
!              number.
!

##      integer function Close_Down(ingerr)
        integer*4 ingerr

##      declare
##      character*256 err_text

##      inquire_ingres (err_text = errortext)
        print *, ' Closing down because of database error:'
        print *, err_text

##      abort
##      exit
        call exit(-1)

        Close_Down = 0
##      end
```

UNIX and VMS—The Employee Query Interactive Forms Application

This EQUEL/FORMS application uses a form in **query** mode to view a subset of the Employee table in the Personnel database. An Ingres query qualification is built at runtime using values entered in fields of the form "empform."

The objects used in this application are:

Object	Description
personnel	The program's database environment.
employee	A table in the database, with six columns: name (**c20**) age (**i1**) idno (**i4**) hired (**date**) dept (**c10**) salary (**money**)
empform	A VIFRED form with fields corresponding in name and type to the columns in the Employee database table. The Name and Idno fields are used to build the query and are the only updatable fields. "Empform" is a compiled form.

A **display** statement drives the application. This statement allows the runtime user to enter values in the two fields that build the query. The **Build_Query** and **Exec_Query** procedures make up the core of the query that is run as a result. Note the way the values of the query operators determine the logic that builds the **where** clause in **Build_Query**. The **retrieve** statement encloses a **submenu** block that allows the user to step through the results of the query.

The retrieved values are not updated, but any employee screen can be saved in a log file using the **printscreen** statement in the **save** menu item.

UNIX

The following **create** statement describes the format of the Employee database table:

```
##    create employee
##        (name     = c20,     /* Employee name */
##         age      = i1,      /* Employee age */
##         idno     = i4,      /* Unique employee id */
##         hired    = date,    /* Date of hire */
##         dept     = c10,     /* Employee department */
##         salary   = money)   /* Annual salary */

C
C   Procedure: MAIN
C   Purpose:   Entry point into Employee Query application.
C
##  program main

##  declare forms
```

```
C           Compiled form
    external empfrm
##  integer*4 empfrm
C               For WHERE clause qualification
    character*100 WhereC

C
C   Initialize global WHERE clause qualification buffer to
C   be an Ingres default qualification that is always true
C
    WhereC = '1=1'

##  forms
##  message 'Accessing Employee Query Application . . .'
##  ingres personnel

##  range of e is employee

##  addform empfrm

##  display 'empfrm' query

##  initialize

##  activate menuitem 'Reset'
##  {
##       clear field all
##  }
##  activate menuitem 'Query'
##  {
C              Verify validity of data
##         validate
           call BldQry(WhereC)
           call ExcQry(WhereC)
##  }

##  activate menuitem 'LastQuery'
##  {
            call ExcQry(WhereC)
##  }

##  activate menuitem 'End', frskey3
##  {
##       breakdisplay
##  }
##  finalize

##  clear screen
##  endforms
##  exit
##  end

C
C   Procedure:  BldQry
C   Purpose:    Build a query from the values in the
C               'name' and 'idno' fields in 'empfrm.'
C   Parameters: WhereC (character string variable to
C               hold WHERE)
C

##  subroutine BldQry(WhereC)

##  declare forms

    character*(*)     WhereC
C            Employee name
```

```
##  character*21      Ename
C           Employee identification number
##  integer*4         Eidno
C           Query operators
##  integer*4         nameop, idop

C   Query operator table maps integer values to string query
    operators   character*2  oprtab(6)
    data oprtab/'=', '!=', '<', '>', '<=', '>='/

##  getform empfrm
##       (Ename = name, nameop = getoper(name),
##        Eidno = idno, idop = getoper(idno))

C   Fill in the WHERE clause

       if ((nameop .EQ. 0) .AND. (idop .EQ. 0)) then
          WhereC = '1=1'
       else
C   User entered a query

          WhereC = ' '

          if ((nameop .NE. 0) .AND. (idop .NE. 0)) then
C   Query on both fields
              write (UNIT=WhereC, FMT=100) oprtab(nameop),
   &                     Ename, oprtab(idop), Eidno
100           format ('e.name', A2, '"', A21, '" and e.idno',
   &                     A2, I6 )

          else if (nameop .NE. 0) then
C
C    Query on the 'name' field. Trailing blanks (A21) not
C    significant because 'name' is type 'C'
C               write (UNIT=WhereC, FMT=110) oprtab(nameop),
   &                     Ename
110             format ('e.name', A2, '"', A21, '"' )

          else
C   Query on the 'idno' field
                write (UNIT=WhereC, FMT=120) oprtab(idop),
&                      Eidno
120             format ('e.idno', A2, I6 )

        endif

      endif

## end

C
C    Procedure:  ExcQry
C    Purpose:    Given a query buffer defining a WHERE clause,
C                issue a RETRIEVE to allow the runtime
C                user to browse the employee
C                found with the given qualification.
C    Parameters: WhereC
C                - Contains WHERE clause qualification.
C
##    subroutine ExcQry(WhereC)

##    declare forms

##    character*(*)       WhereC
C           Matches Employee table
C           Employee Name
```

```
##    character*21        Ename
C             Employee Age
##    integer*2           Eage
C             Employee Identification Number
##    integer*4           Eidno
C             Employee Hire Date
##    character*26        Ehired
C             Employee Department
##    character*11        Edept
C             Employee Salary
##    real*4              Epay
C             Flag, were any rows found ?
##    integer*4           rows

##    retrieve (Ename = e.name, Eage = e.age, Eidno = e.idno,
##            Ehired = e.hired, Edept = e.dept, Epay = e.salary)
##            where WhereC
##    {
C     Put values on to form and display them
##    putform empfrm
##         (name = Ename, age = Eage, idno = Eidno, hired = Ehired,
##          dept = Edept, salary = Epay)
##    redisplay
##    submenu
##    activate menuitem 'Next', frskey4
##    {
C
C     Do nothing, and continue with the RETRIEVE loop. The
C     last one will drop out.
C
##      }

##      activate menuitem 'Save', frskey8
##      {
C     Save screen data in log file
##            printscreen (file = 'query.log')
C     Drop through to next employee
##      }

##    activate menuitem 'End', frskey3
##      {
C     Terminate the RETRIEVE loop
##        endretrieve
##      }
##    }

##    inquire_equel (rows = ROWCOUNT)
      if (rows .EQ. 0) then
##       message 'No rows found for this query'
      else
##       clear field all
##       message 'No more rows. Reset for next query'
      endif

##    sleep 2

##    end
```

VMS

The following **create** statement describes the format of the Employee database table:

```
##      create employee
##          (name     = c20,    /* Employee name */
##           age      = i1,     /* Employee age */
```

```
##           idno     = i4,     /* Unique employee id */
##           hired    = date,   /* Date of hire */
##           dept     = c10,    /* Employee department */
##           salary   = money)  /* Annual salary */

!
!     Procedure:   MAIN
!     Purpose:     Entry point into Employee Query application.
!
##    program main

##    declare forms


      external empfrm        ! Compiled form
##    integer*4 empfrm
      character*100 WhereC   ! For WHERE clause qualification
      !
      ! Initialize global WHERE clause qualification buffer to
      ! be an Ingres default qualification that is
      ! always true
      !
      WhereC = '1=1'

##    forms
##    message 'Accessing Employee Query Application . . .'
##    ingres personnel

##    range of e is employee

##    addform empfrm

##    display 'empfrm' query
##    initialize

##    activate menuitem 'Reset'
##    {
##          clear field all
##    }

##    activate menuitem 'Query'
##    {
            ! Verify validity of data
##          validate
            call BldQry(WhereC)
            call ExcQry(WhereC)
##    }

##    activate menuitem 'LastQuery'
##    {
            call ExcQry(WhereC)
##    }

##    activate menuitem 'End', frskey3
##    {
##          breakdisplay
##    }
##    finalize

##    clear screen
##    endforms
##    exit
##    end

!
!    Procedure:    BldQry
```

```
!    Purpose:      Build a query from the values in the
!                  'name' and 'idno' fields in 'empfrm.'
!    Parameters:   WhereC (character string variable to
!                  hold WHERE)
!

##    subroutine BldQry(WhereC)

##    declare forms

      character*(*)  WhereC
##    character*21   Ename          ! Employee name
##    integer*4      Eidno        ! Employee identification number
##    integer*4      nameop, idop   ! Query operators

      ! Query operator table maps integer values to string query
      ! operators
      character*2 oprtab(6)
      data oprtab/'=', '!=', '<', '>', '<=', '>='/

##    getform empfrm
##         (Ename = name, nameop = getoper(name),
##          Eidno = idno, idop = getoper(idno))

      ! Fill in the WHERE clause

      if ((nameop .EQ. 0) .AND. (idop .EQ. 0)) then
          WhereC = '1=1'
      else
            ! User entered a query
            WhereC = ' '

            if ((nameop .NE. 0) .AND. (idop .NE. 0)) then
                ! Query on both fields
                write (UNIT=WhereC, FMT=100) oprtab(nameop),
     1               Ename, oprtab(idop), Eidno
100             format ('e.name', A2, '"', A21, '" and e.idno',
     1               A2, I6 )

            else if (nameop .NE. 0) then
                !
                ! Query on the 'name' field. Trailing blanks
                ! (A21) not significant because 'name' is type
                ! 'C'
                write (UNIT=WhereC, FMT=110) oprtab(nameop),
     1               Ename
110             format ('e.name', A2, '"', A21, '"' )

            else
                ! Query on the 'idno' field
                write (UNIT=WhereC, FMT=120) oprtab(idop),
     1                 Eidno
120             format ('e.idno', A2, I6 )

            endif
      endif

##    end

!
!     Procedure:  ExcQry
!     Purpose:    Given a query buffer defining a WHERE clause,
!                 issue a RETRIEVE to allow the runtime user to
!                 browse the employee found with the given
!                 qualification.
!     Parameters: WhereC
```

```
!                - Contains WHERE clause qualification.
!
##    subroutine ExcQry(WhereC)

##    declare forms
##    character*(*) WhereC ! Matches Employee table
##    character*21  Ename  ! Employee Name
      integer*2     Eage   ! Employee Age
##    integer*4     Eidno  ! Employee Identification Number
##    character*26  Ehired ! Employee Hire Date
##    character*11  Edept  ! Employee Department
##    real*4        Epay   ! Employee Salary
##    integer*4     rows   ! Flag, were any rows found ?

##    retrieve (Ename = e.name, Eage = e.age, Eidno = e.idno,
##            Ehired = e.hired, Edept = e.dept, Epay = e.salary)
##            where WhereC
##    {
##            ! Put values on to form and display them
##            putform empfrm
##              (name = Ename, age = Eage, idno = Eidno, hired =
##               Ehired, dept = Edept, salary = Epay)
##            redisplay
##            submenu
##            activate menuitem 'Next', frskey4
##            {
                !
                ! Do nothing, and continue with the RETRIEVE
                ! loop. The last one will drop out.
                !
##            }

##            activate menuitem 'Save', frskey8
##            {
                ! Save screen data in log file
##              printscreen (file = 'query.log')
                ! Drop through to next employee
##            }

##            activate menuitem 'End', frskey3
##           {
                ! Terminate the RETRIEVE loop
##          endretrieve
##        }
##

##    inquire_equel (rows = ROWCOUNT)
      if (rows .EQ. 0) then
##            message 'No rows found for this query'
      else
##            clear field all
##            message 'No more rows. Reset for next query'
      endif

##    sleep 2

##    end
```

UNIX and VMS—The Table Editor Table Field Application

This EQUEL/FORMS application uses a table field to edit the Person table in the Personnel database. It allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate their use and their interaction with an Ingres database.

The application uses the following objects:

Object	Description
personnel	The program's database environment.
person	A table in the database with three columns: name (**c20**) age (**i2**) number (**i4**)
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form with two columns: name (**c20**) age (**i4**) When initialized, the table field includes the hidden number (**i4**) column.

At the beginning of the application, a **retrieve** statement is issued to load the table field with data from the Person table. After loading the table field, you can browse and edit the displayed values. You can add, update, or delete entries. When finished, the values are unloaded from the table field, and, in a multi-statement transaction, your updates are transferred back into the Person table.

The application runs in the UNIX and VMS environments.

UNIX

The following **create** statement describes the format of the Person database table:

```
##    create person
##        (name    = c20   /* Person name */
##         age     = i2,   /* Age */
##         number  = i4)   /* Unique id number */

C
C    Procedure: MAIN
C    Purpose: Entry point into Table Editor program.
C
##   program main

C    Table field row states
```

```
C              Empty or undefine row
     parameter (stUDEF=0)
C              Appended by user
     parameter (stNEW=1)
C              Loaded by program - not updated
     parameter (stUCHG=2)
C              Loaded by program - since changed
     parameter (stCHG=3)
C              Deleted by program
     parameter (stDEL=4)

##   declare forms
C    Table field entry information
C              State of date set entry
##   integer*4 state
C              Record number
##   integer*4 row
C              Last row in table field
##   integer*4 lstrow

C    Utility buffers
C              Message buffer
##   character*256 msgbuf
C              Response buffer
##   character*20 rspbuf

C    Status variables
C              Update error from database
##   integer*4 upderr
C              Number of rows updated
##   integer*4 updrow
C              Transaction aborted
     logical xactq
C              Save changes for quit
     logical savchg

C    The following variables correspond to the 'person' table
C              Full name
##   character*20 pname
C              Age of person
##   integer*4               page
C              Unique person number
##   integer*4               pnum
C              Max person id
##   integer*4               maxid

C    Start up Ingres and the FORMS system
##   INGRES 'personnel'

##   forms
C    Verify that the user can edit the 'person' table
##   prompt noecho ('Password for table editor: ', rspbuf)

     if ( rspbuf .NE. 'MASTER_OF_ALL') then
##        message 'No permission for task. Exiting . . .'
##        endforms
##        exit
          call exit(-1)
     endif

##   message 'Initializing Person Form . . .'

##   range of p IS person

##   forminit person
C
```

```
C    Initialize 'persontbl' table field with a data set in FILL
C    mode so that the runtime user can append rows. To keep track
C    of events occurring to original rows that will be loaded into
C    the table field, hide the unique person number.
C
##   inittable person persontbl fill (number = integer)
     call LdTab(pers)

##   display person update
##   initialize

##   activate menuitem 'Top', frskey5
##   {
C
C         Provide menu, as well as the system FRS key to scroll
C         to both extremes of the table field.
C
##        scroll person persontbl TO 1
##   }

##   activate menuitem 'Bottom', frskey6
##   {
##        scroll person persontbl to end  /* Forward */
##   }

##   activate menuitem 'Remove'
##   {
C
C         Remove the person in the row the user's cursor is on.
C         If there are no persons, exit operation with message.
C         Note that this check cannot really happen as there is
C         always an UNDEFINED row in FILL mode.
C
##        inquire_frs table person (lstrow = lastrow(persontbl))

        if (lstrow .EQ. 0) then
##            message 'Nobody to Remove'
##            sleep 2
##            resume field persontbl
        endif

##      deleterow person persontbl /* Record later */
##   }

##   activate menuitem 'Find', frskey7
##   {
C
C        Scroll user to the requested table field entry.
C        Prompt the user for a name, and if one is typed in
C        loop through the data set searching for it.
C
##       prompt ('Enter name of person: ', rspbuf)
         if (rspbuf .EQ. ' ') then
##           resume field persontbl
         endif

##       unloadtable person persontbl
##           (pname = name, row = _RECORD, state = _STATE)
##       {
C    Do not compare with deleted rows
         if ( (pname .EQ. rspbuf) .AND.
&             (state .NE. stDEL) ) then
##            scroll person persontbl to row
##            resume field persontbl
         endif
```

```
##     }

C    Fell out of loop without finding name
           msgbuf = 'Person "' // rspbuf //
&                   '" not found in table [HIT RETURN]'
##         prompt noecho (msgbuf, rspbuf)
##     }

##   activate menuitem 'Save', frskey8
##   {
##         validate field persontbl
           savchg = .TRUE.
##         breakdisplay
##   }

##   activate menuitem 'Quit', frskey2
##   {
           savchg = .FALSE.
##         breakdisplay
##   }
##   finalize

       if ( .NOT. savchg ) then
##          endforms
##          exit
            call exit(1)
       endif

C
C    Exit person table editor and unload the table field. If any
C    updates, deletions or additions were made, duplicate these
C    changes in the source table. If the user added new people we
C    must assign a unique person id before returning it to the
C    table. To do this, increment the previously saved maximum
C    id number with each insert.
C
C    Do all the updates in a transaction (for simplicity,
C    this transaction does not restart on DEADLOCK error: 4700)
C
##   begin transaction

     upderr = 0
     xactq = .FALSE.

##   message 'Exiting Person Application . . .';

##   unloadtable person persontbl
##          (pname = name, page = age,
##          pnum = number, state = _STATE)
##   {
           if (state .EQ. stNEW) then
C    Appended by user. Insert with new unique id
                maxid = maxid + 1
##              repeat append to person (name = @pname,
##                                       age = @page,
##                                       number = @maxid)
           else if (state .EQ. stCHG) then
C    Updated by user. Reflect in table
##             repeat replace p (name = @pname, age = @page)
##                    where p.number = @pnum
           else if (state .EQ. stDEL) then
C
C               Deleted by user, so delete from table. Note that
C               only original rows are saved by the program, and
C               not rows appended at runtime.
C
```

```
##              repeat delete from p where p.number = @pnum
            else
C               state .EQ. UNCHANGED or UNDEFINED - No updates
            endif

C
C    Handle error conditions -
C    If an error occurred, then ABORT the transaction.
C    If no rows were updated then inform user, and
C    prompt for continuation.
C
##      inquire_ingres (upderr = ERRORNO, updrow = ROWCOUNT)

          if (upderr .GT. 0) then
##            inquire_equel (msgbuf = errortext)
##            abort
              xactq = .true.
##            endloop
          else if (updrow .EQ. 0) then
              msgbuf = 'Person "' // pname //
&                      '" not updated. Abort all updates? '
##            prompt (msgbuf, rspbuf)
              if ((rspbuf(1:1) .EQ. 'Y') .OR.
&                 (rspbuf(1:1) .EQ. 'y')) then
##                abort
                  xactq = .TRUE.
##                endloop
               endif
           endif

##         }     /* end of UNLOADTABLE loop */

     if ( .NOT. xactq ) then
##             end transaction  /* Commit the updates */
     endif

##   endforms           /* Terminate the FORMS and Ingres */
##   exit

     if (upderr .NE. 0) then
           print *, 'Your updates were aborted because of error: '
           print *, msgbuf
     endif
C    end of main
##   end

C
C    Subroutine:    LdTab
C    Purpose:       Load the table field from the 'person' table.
C                   The columns 'name' and 'age' will be displayed,
C                   and 'number' will be hidden.
C Parameters:
C                   None
C   Returns:
C                   Nothing
C
##   subroutine LdTab(pers)
C    Set up error handling for loading procedure
##   declare forms

C    The following variables correspond to the 'person' table
C            Full name
##       character*20 pname
C            Age of person
##       integer*4 page
C            Unique person number
```

```
##       integer*4 pnum
C           Max person id number
##       integer*4 maxid

##   message 'Loading Person Information . . .'
C
C    Fetch the maximum person id number for later use
C    PERFORMANCE NOTE: max() will do sequential scan of table
C
##   retrieve (maxid = max(p.number))

C    Fetch data, and load table field
##   retrieve (pname = p.name, page = p.age,
##             pnum = p.number)
##   {
##         loadtable person persontbl
##             (name = pname, age = page,
##             number = pnum)
##   }
##   end
```

VMS

The following **create** statement describes the format of the Person database table:

```
##      create person
##                (name    = c20,      /* Person name */
##                 age     = i2,       /* Age */
##                 number  = i4)       /* Unique id number */

!
!   Procedure: MAIN
!   Purpose: Entry point into Table Editor program.
!
##  program main

    ! Table field row states
    parameter (stUNDEF=0)       ! Empty or undefined row
    parameter (stNEW=1)         ! Appended by user
    parameter (stUNCHANGED=2)   ! Loaded by program - not updated
    parameter (stCHANGE=3)      ! Loaded by program - since changed
    parameter (stDELETE=4)     ! Deleted by program

##  declare forms
                                  ! Table field entry information
##  integer*4 state                ! State of data set entry
##  integer*4 row                  ! Record number
##  integer*4 lastrow             ! Last row in table field

    ! Utility buffers
##  character*256 msgbuf           ! Message buffer
##  character*20 respbuf           ! Response buffer

    ! Status variables
##  integer*4 update_error         ! Update error from database
##  integer*4 update_rows          ! Number of rows updated
    logical xact_aborted           ! Transaction aborted
    logical save_changes           ! Save changes or Quit

    ! structure person corresponds to 'person' table
##  structure /person/
##             character*20 pname   ! Full name
##             integer*4 page       ! Age of person
##             integer*4 pnumber    ! Unique person number
##             integer*4 maxid      ! Max person id number
```

```
##  end structure
##  record /person/ pers

    ! Start up Ingres and the FORMS system
##  ingres 'personnel'

##  forms

    ! Verify that the user can edit the 'person' table
##  prompt noecho ('Password for table editor: ', respbuf)

    if ( respbuf .NE. 'MASTER_OF_ALL') then
##        message 'No permission for task. Exiting . . .'
##        endforms
##        exit
          call exit(-1)
    endif

##  message 'Initializing Person Form . . .'

##  range of p is person

##  forminit personfrm

    !
    !  Initialize 'persontbl' table field with a data set in
    !  FILL mode so that the runtime user can append rows.
    !  To keep track of events occurring to original rows that
    !  will be loaded into the table field, hide the unique
    !  person number.
    !
##  inittable personfrm persontbl fill (number = integer)

    call Load_Table(pers)

##  display personfrm update
##  initialize

##  activate menuitem 'Top', frskey5
##  {
            !
            ! Provide menu, as well as the system FRS key to scroll
            ! to both extremes of the table field.
            !
##           scroll personfrm persontbl to 1
##  }

##  activate menuitem 'Bottom', frskey6
##  {
##           scroll personfrm persontbl to end     /* Forward */
##  }

##  activate menuitem 'Remove'
##  {
         !
         ! Remove the person in the row the user's cursor is on.
         ! If there are no persons, exit operation with message.
         ! Note that this check cannot really happen as there is
         ! always an UNDEFINED row in FILL mode.
         !
##       inquire_frs table personfrm (lastrow = lastrow(persontbl))

           if (lastrow .EQ. 0) then
##             message 'Nobody to Remove'
##             sleep 2
##             resume field persontbl
```

```
           endif

##         deleterow personfrm persontbl         /* Record later */
##  }
##  activate menuitem 'Find', frskey7
##  {
           !
           ! Scroll user to the requested table field entry.
           ! Prompt the user for a name, and if one is typed in
           ! loop through the data set searching for it.
           !
##         prompt ('Enter name of person: ', respbuf)
           if (respbuf .EQ. ' ') then
##             resume field persontbl
           endif

##         unloadtable personfrm persontbl
##           (pers.pname = name, row = _RECORD, state = _STATE)
##         {
             ! Do not compare with deleted rows
             if ( (pers.pname .EQ. respbuf) .AND.
     1            (state .NE. stDELETE) ) then
##               Scroll personfrm persontbl to row
##               resume field persontbl
             endif

##         }

           ! Fell out of loop without finding name
           msgbuf = 'Person "' // respbuf //
     1              '" not found in table [HIT RETURN]'
##         prompt noecho (msgbuf, respbuf)
##         }

##         activate menuitem 'Save', frskey8
##         {
##               validate field persontbl
                 save_changes = .TRUE.
##               breakdisplay
##         }

##         activate menuitem 'Quit', frskey2
##         {
                 save_changes = .FALSE.
##               breakdisplay
##         }
##         finalize

           if ( save_changes .NE. .TRUE.) then
##              endforms
##              exit
                call exit(1)
           endif

     !
     ! Exit person table editor and unload the table field. If any
     ! updates, deletions or additions were made, duplicate these
     ! changes in the source table. If the user added new people we
     ! must assign a unique person id before returning it to
     ! the table. To do this, increment the previously saved
     ! maximum id number with each insert.
     !
     ! Do all the updates in a transaction (for simplicity,
     ! this transaction does not restart on DEADLOCK error: 4700)
     !
##   begin transaction
```

```
     update_error = 0
     xact_aborted = .FALSE.

##   message 'Exiting Person Application . . .';

##   unloadtable personfrm persontbl
##     (pers.pname = name, pers.page = age,
##      pers.pnumber = number, state = _STATE)
##   {
        if (state .EQ. stNEW) then
            ! Appended by user. Insert with new unique id
            pers.maxid = pers.maxid + 1
##          repeat append to person    (name = @pers.pname,
##                                      age = @pers.page,
##                                      number = @pers.maxid)
        else if (state .EQ. stCHANGE) then
            ! Updated by user. Reflect in table
##          repeat replace p (name = @pers.pname, age = @pers.page)
##                   where p.number = @pers.pnumber
        else if (state .EQ. stDELETE) then
            !
            ! Deleted by user, so delete from table. Note that only
            ! original rows are saved by the program, and not rows
            ! appended at runtime.
            !
##          repeat delete from p where p.number = @pers.pnumber
        else
             ! state .EQ. UNCHANGED or UNDEFINED - No updates
        endif

        !
        ! Handle error conditions -
        ! If an error occurred, then ABORT the transaction.
        ! If no rows were updated then inform user, and
        ! prompt for continuation.
        !
##      inquire_ingres (update_error = ERRORNO,
##            update_rows = ROWCOUNT)

        if (update_error .GT. 0) then              ! Error
##          inquire_equel (msgbuf = ERRORTEXT)
##          abort
            xact_aborted = .TRUE.
##          endloop
        else if (update_rows .EQ. 0) then
            msgbuf = 'Person "' // pers.pname //
    1                '" not updated. Abort all updates? '
##          prompt (msgbuf, respbuf)
            if  ((respbuf(1:1) .EQ. 'Y') .OR.
    1            (respbuf(1:1) .EQ. 'y')) then
##              abort
                xact_aborted = .TRUE.
##             endloop
            endif
        endif

##    }         /* end of UNLOADTABLE loop */

    if (xact_aborted .EQ. .FALSE.) then
##       end transaction        /* Commit the updates */
    endif

##  endforms            /* Terminate the FORMS and Ingres */
##  exit
```

```
    if (update_error .NE. 0) then
         print *, 'Your updates were aborted because of error: '
         print *, msgbuf
    endif
##  end               ! Main Program

!
!   Subroutine: Load_Table
!   Purpose:    Load the table field from the 'person' table. The
!               columns 'name' and 'age' will be displayed, and
!               'number' will be hidden.
!   Parameters:
!               None
!   Returns:
!               Nothing
!
##  subroutine Load_Table(pers)
    ! Set up error handling for loading procedure
##  declare forms

    ! structure person corresponds to 'person' table
##  structure /person/
##              character*20 pname     ! Full name
##              integer*4 page         ! Age of person
##              integer*4 pnumber      ! Unique person number
##              integer*4 maxid        ! Max person id number
##  end structure
##  record /person/ pers

##  message 'Loading Person Information . . .'
    !
    ! Fetch the maximum person id number for later use
    ! PERFORMANCE NOTE: max() will do sequential scan of table
    !
##  retrieve (pers.maxid = max(p.number))

    ! Fetch data, and load table field
##  retrieve (pers.pname = p.name, pers.page = p.age,
##                  pers.pnumber = p.number)
##       {
##           loadtable personfrm persontbl
##             (name = pers.pname, age = pers.page,
##              number = pers.pnumber)
##       }

##  end
```

UNIX and VMS—The Professor-Student Mixed Form Application

This EQUEL/FORMS application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The application uses the following objects:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**c25**) pdept (**c10**) See its create statement below for a full description.
student	A database table with seven columns: sname (**c25**) sage (**i1**) sbdate (**c25**) sgpa (**f4**) sidno (**i1**) scomment (text(**200**)) sadvisor (**c25**) See the create statement below for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has fields pname and pdept, which correspond to the information in the Professor table and the Studenttbl table field. The pdept field is display-only. "Masterfrm" is a compiled form.
studenttbl	A table field in "masterfrm" with two columns, "sname" and "sage." When initialized, it also has five more hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the Sgpa, Scomment, and Sadvisor fields are updatable. "Studentfrm" is a compiled form.
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the "studentfrm" form and the Studenttbl table field.

The program uses the "masterfrm" as the general-level master entry, in which the user can only retrieve and browse data, and the "studentfrm" as the detailed screen, in which the user can update specific student information.

The user enters a name in the pname (professor name) field and then selects the **Students** menu operation. The operation fills the displayed and hidden columns of the studenttbl table field with detailed information about the students of the named professor. The user may then browse the table field (in **read** mode), which displays only the names and ages of the students. To request more information about a specific student, the user can select the **Zoom** menu operation. This operation displays the form "studentfrm." The fields of "studentfrm" are filled with values stored in the hidden columns of studenttbl. The user can make changes to three fields (sgpa, scomment and sadvisor). If validated, these changes are written back to the database table (based on the unique student ID), and to the table field's data set. The user can repeat this process for different professor names.

The application runs on UNIX and VMS.

UNIX

The following two **create** statements describe the Professor and Student database tables:

```
##  create student            /* Graduate student table */
##    (sname     = c25,       /* Name */
##     sage      = i1,        /* Age */
##     sbdate    = c25,       /* Birth date */
##     sgpa      = f4,        /* Grade point average */
##     sidno     = i4,        /* Unique student number */
##     scomment  = text(200), /* General comments */
##     sadvisor  = c25)       /* Advisor's name */

##  create professor          /* Professor table */
##    (pname     = c25,       /* Professor's name */
##     pdept     = c10)       /* Department */

C
C    Procedure:      MAIN
C    Purpose:        Start up program and call Master driver.
C
##   program main

C    Start up Ingres and the FORMS system
##   declare forms

##   forms
##   message 'Initializing Student Administrator . . .'

##   INGRES personnel
##   range of p IS professor, s IS student

     call Master()

##   clear screen
##   endforms

##   exit

##   end
```

```
C
C    Procedure:  Master
C    Purpose:    Drive the application, by running 'mstfrm', and
C                allowing the user to 'zoom' into a
C                selected student.
C    Parameters:
C                None - Uses the global student 'grad' record.
C

##   subroutine Master()
##   declare forms

C        Declare function
     logical StdChg

C    grad student record maps to database table
C           Student's name
##   character*25     Sname
C           Student's age
##   integer*2        Sage
C           Student's birthday
##   character*25     Sbdate
C           Student's grade point average
##   real*4           Sgpa
C           Student's unique id number
##   integer*4        Sidno
C           General comment field
##   character*200    Scomm
C           Student's advisor
##   character*25     Sadv

C    Professor info maps to database table
C           Professor's name
##   character*25     Pname
C           Professor's department
##   character*10     Pdept

C    Useful forms system information
C           Last row in table field
##   integer*4        lstrow
C           Is it a table field ?
##   integer*4        istab

C    Local utility buffers
C           Message buffer
##   character*100    msgbuf
C           Response buffer
##   character*256    rspbuf
C           Old advisor before ZOOM
##   character*25     oldadv

C    Externally compiled master form
     external         mstfrm
##   integer*4        mstfrm

##   addform mstfrm

C
C    Initialize 'studenttbl' with a data set in READ mode.
C    Declare hidden columns for all the extra fields that
C    the program will display when more information is
C    requested about a student. Columns 'sname' and 'sage'
C    are displayed, all other columns are hidden, to be
C    used in the student information form.
C
##   inittable #mstfrm studenttbl read
```

```
##        (#sbdate    = c25,
##         #sgpa      = f4,
##         #sidno     = i4,
##         #scomment  = text(200),
##         #sadvisor  = c20)

##   display #mstfrm update

##    initialize
##    {
##       message 'Enter an Advisor name . . .'
##       sleep 2
##    }

##    activate menuitem 'Students', field 'pname'
##    {
C        Load the students of the specified professor
         Pname = ' '
##       getform (Pname = #pname)

C        If no professor name is given then resume
         if (Pname .EQ. ' ') then
##       resume field #pname
         endif

C
C       Verify that the professor exists.   Local error
C       handling just prints the message, and continues.
C       We assume that each professor has exactly one
C       department.
C
      Pdept = ' '
##    retrieve (Pdept = p.#pdept)
##         where p.#pname = Pname

      if (Pdept .EQ. ' ') then
          msgbuf = 'No professor with name "' // Pname //
   &               '"[RETURN]'
##        prompt noecho (msgbuf, rspbuf)
##        clear field all
##        resume field #pname
      endif

C     Fill the department field and load students
##    putform (#pdept = Pdept)
##    redisplay                /* Refresh for query */
      call LdStd (Pname)

##    resume field studenttbl

##  }               /* 'Students' */

##  activate menuitem 'Zoom'
##  {

C
C     Confirm that user is on 'studenttbl', and that
C     the table field is not empty. Collect data from
C     the row and zoom for browsing and updating.
C
##    inquire_frs field #mstfrm (istab = table)

    if (istab .EQ. 0) then
##      prompt noecho
##           ('Select from the student table [RETURN]', rspbuf)
##      resume field studenttbl
```

```
    endif

##  inquire_frs table #mstfrm (lstrow = lastrow)

    if (lstrow .EQ. 0) then
##      prompt noecho ('There are no students [RETURN]', rspbuf)
##      resume field #pname
    endif

C   Collect all data on student into global record
##  getrow #mstfrm studenttbl
##      (Sname     = #sname,
##       Sage      = #sage,
##       Sbdate    = #sbdate,
##       Sgpa      = #sgpa,
##       Sidno     = #sidno,
##       Scomm     = #scomment,
##       Sadv      = #sadvisor)

C
C   Display 'stdfrm', and if any changes were made
C   make the updates to the local table field row.
C   Only make updates to the columns corresponding to
C   writable fields in 'stdfrm'.   If the student
C   changed advisors, then delete this row from the
C   display.
C
    oldadv = Sadv
    if (StdChg (Sname, Sage, Sbdate, Sgpa, Sidno, Scomm,
  &      Sadv)) then
        if (oldadv .NE. Sadv) then
##          deleterow #mstfrm studenttbl
        else
##         putrow #mstfrm studenttbl
##           (#sgpa       = Sgpa,
##           <x:c3>#scomment   = Scomm,
##           #sadvisor    = Sadv)
        endif
       endif

##   }                 /* 'Zoom' */

##    activate menuitem 'Quit', frskey2
##    {
##       breakdisplay
##    }               /* 'Quit' */

##    finalize

##    end

C
C   Procedure: LdStd
C   Purpose:   Given an advisor name, load into the 'studenttbl'
C              table field all the students who report to the
C              professor with that name.
C   Parameters:
C              advisor - User specified professor name.
C              Uses the global student record.
C

##  subroutine LdStd(advisor)

##  declare forms

##  character*(*)  advisor
```

```
C   grad student record maps to database table
C         Student's name
##  character*25     Sname
C         Student's age
##  integer*2        Sage
C        Student's birthday
##  character*25     Sbdate
C         Student's grade point average
##  real*4           Sgpa
C        Student's unique id number
##  integer*4        Sidno
C        General comment field
##  character*200    Scomm
C         Student's advisor
##  character*25     Sadv

C
C   Clear previous contents of table field. Load the table
C   field from the database table based on the advisor name.
C   Columns 'sname' and 'sage' will be displayed, and all
C   others will be hidden.
C
##  message 'Retrieving Student Information . . .'
##  clear field studenttbl

##  retrieve
##         (Sname     = s.#sname,
##          Sage      = s.#sage,
##          Sbdate    = s.#sbdate,
##          Sgpa      = s.#sgpa,
##          Sidno     = s.#sidno,
##          Scomm     = s.#scomment,
##          Sadv      = s.#sadvisor)
##          where s.sadvisor          = advisor
##  {
##          loadtable #mstfrm studenttbl
##            (#sname     = Sname,
##              #sage     = Sage,
##              #sbdate   = Sbdate,
##              #sgpa     = Sgpa,
##              #sidno    = Sidno,
##              #scomment = Scomm,
##              #sadvisor = Sadv)

##  }

##  end
C
C   Procedure: StdChg
C   Purpose:   Allow the user to zoom into the details of a
C              selected student. Some of the data can be updated
C              by the user. If any updates were made, then
C              reflect these back into the database table.
C              The procedure returns TRUE if any changes were made.
C   Parameters:
C              None - Uses with data in the global 'grad' record.
C   Returns:
C              TRUE/FALSE - Changes were made to the database.
C              Sets the global 'grad' record with the new data.
C

##  logical function StdChg(Sname, Sage, Sbdate, Sgpa, Sidno,
##&                         Scomm, Sadv)
##  declare forms

C   grad student record maps to database table
```

```
C         Student's  name
##  character*25     Sname
C         Student's age
##  integer*2        Sage
C         Student's birthday
##  character*25     Sbdate
C         Student's grade point average
##  real*4           Sgpa
C         Student's unique id number
##  integer*4        Sidno
C         General comment field
##  character*200    Scomm
C         Student's advisor
##  character*25     Sadv

C   Changes made to date in form
##  integer*4        chnged
C         Valid advisor name ?
##  integer*4        vldadv

C         Compiled form
    external         stdfrm
##  integer*4        stdfrm

C   Control ADDFORM to only initialize once
    integer*4     ldform
    data          ldform/0/

    if (ldform .EQ. 0) then
##      message 'Loading Student form . . .'
##      addform stdfrm
        ldform = 1
    endif

##  display #Stdfrm fill
##  initialize
##       (#sname       = Sname,
##        #sage       = Sage,
##        #sbdate       = Sbdate,
##        #sgpa       = Sgpa,
##        #sidno      = Sidno,
##        #scomment      = Scomm,
##        #sadvisor      = Sadv)

##  activate menuitem 'Write'
##  {
C
C      If changes were made then update the database
C      table. Only bother with the fields that are not
C      read-only.
C
##     inquire_frs form (chnged = change)

       if (chnged .EQ. 1) then
##         validate

##         message 'Writing changes to database. . .'
##         getform
##            (Sgpa    = #sgpa,
##             Scomm   = #scomment,
##             Sadv    = #sadvisor)

C          Enforce integrity of professor name
           vldadv = 0
##         retrieve (vldadv = 1)
##              where p.pname = Sadv
```

```
           if (vldadv .EQ. 0) then
##             message 'Not a valid advisor name'
##             sleep 2
##             resume field sadvisor
           else
##             replace s (#sgpa = Sgpa,
##                        scomment = Scomm,
##                        sadvisor = Sadv)
##             where s.#sidno = Sidno
##             breakdisplay
             endif
          endif
##      }
                        /* 'Write' */

##      activate menuitem 'End', frskey3
##      {
C           Quit without submitting changes
            chnged = 0
##          breakdisplay
##      }
                     /* 'Quit' */

##      finalize

        if (chnged .EQ. 1) then
            StdChg = .TRUE.
        else
            StdChg = .FALSE.
        endif

        return

##      end
```

VMS

The following two **create** statements describe the professor and student database tables:

```
##    create student                /* Graduate student table */
##           (sname    = c25,       /* Name */
##            sage     = i1,        /* Age */
##            sbdate   = c25,       /* Birth date */
##            sgpa     = f4,        /* Grade point average */
##            sidno    = i4,        /* Unique student number */
##            scomment = text(200), /* General comments */
##            sadvisor = c25)       /* Advisor's name */

##    create professor              /* Professor table */
##           (pname    = c25,       /* Professor's name */
##            pdept    = c10)       /* Department */

!
!       Procedure: MAIN
!       Purpose:   Start up program and call Master driver.
!
##      program main

        ! Start up Ingres and the FORMS system
##      declare forms

##      forms
##      message 'Initializing Student Administrator . . .'
```

```
##      ingres personnel
##      range of p IS professor, s IS student

        call Master()
##      clear screen
##      endforms
##      exit

##      end

!
!       Procedure: Master
!       Purpose:   Drive the application, by running 'masterfrm',
!                  and allowing the user to 'zoom' into a
!                  selected student.
!       Parameters:
!                  None - Uses the global student 'grad' record.
!

##      subroutine Master()
##      declare forms

        logical Student_Info_Changed ! function

        ! grad student record maps to database table
##      structure /grad_student/
##          character*25    sname
##          integer*2       sage
##          character*25    sbdate
##          real*4          sgpa
##          integer*4       sidno
##          character*200   scomment
##          character*25    sadvisor
##      end structure
##      record /grad_student/ grad

        ! Professor info maps to database table
##      structure /professor/
##          character*25 pname
##          character*10 pdept
##      end structure
##      record /professor/ prof

        ! Useful forms system information
##      integer*4 lastrow ! Lastrow in table field
##      integer*4 istable ! Is a table field?

        ! Local utility buffers
##      character*100 msgbuf     ! Message buffer
##      character*256 respbuf    ! Response buffer
##      character*25 old_advisor ! Old advisor before ZOOM

        ! Externally compiled master form
        external masterfrm
##      integer*4 masterfrm

##      addform masterfrm

        !
        ! Initialize 'studenttbl' with a data set in READ mode.
        ! Declare hidden columns for all the extra fields that
        ! the program will display when more information is
        ! requested about a student. Columns 'sname' and 'sage'
        ! are displayed, all other columns are hidden, to be
        ! used in the student information form.
        !
```

```
##      inittable #masterfrm studenttbl read
##           (sbdate   = c25,
##            sgpa     = float,
##            sidno    = integer,
##            scomment = c200,
##            sadvisor = c20)

##      display #masterfrm update

##      initialize
##      {
##          message 'Enter an Advisor name . . .'
##          sleep 2
##      }

##      activate menuitem 'Students', field 'pname'
##      {
            ! Load the students of the specified professor
##          getform (prof.pname = pname)

            ! If no professor name is given then resume
            if (prof.pname .EQ. ' ') then
##              resume field pname
            endif

            !
            ! Verify that the professor exists. Local error
            ! handling just prints the message, and continues.
            ! We assume that each professor has exactly one
            ! department.
            !

            prof.pdept = ' '
##          retrieve (prof.pdept = p.pdept)
##               where p.pname = prof.pname

            if (prof.pdept .EQ. ' ') then
            msgbuf = 'No professor with name "' // prof.pname //
     1                    '"[RETURN]'
##               prompt noecho (msgbuf, respbuf)
##               clear field all
##               resume field pname
            endif

            ! Fill the department field and load students
##          putform (pdept = prof.pdept)
##          redisplay /* Refresh for query */

            call Load_Students(prof.pname)

##          resume field studenttbl

##      }              /* 'Students' */

##      activate menuitem 'Zoom'
##      {

        !
        ! Confirm that user is on 'studenttbl', and that
        ! the table field is not empty. Collect data from
        ! the row and zoom for browsing and updating.
        !
##      inquire_frs field #masterfrm (istable = table)

        if (istable .EQ. 0) then
```

```
##            prompt noecho
##             ('Select from the student table [RETURN]', respbuf)
##            resume field studenttbl
        endif

##      inquire_frs table #masterfrm (lastrow = lastrow)
        if (lastrow .EQ. 0) then
##       prompt noecho ('There are no students [RETURN]', respbuf)
##       resume field pname
        endif

        ! Collect all data on student into global record
##      getrow #masterfrm studenttbl
##          (grad.sname     = sname,
##           grad.sage      = sage,
##           grad.sbdate    = sbdate,
##           grad.sgpa      = sgpa,
##           grad.sidno     = sidno,
##           grad.scomment  = scomment,
##           grad.sadvisor  = sadvisor)

        !
        ! Display 'studentfrm', and if any changes were made
        ! make the updates to the local table field row.
        ! Only make updates to the columns corresponding to
        ! writable fields in 'studentfrm'. If the student
        ! changed advisors, then delete this row from the
        ! display.
        !

        old_advisor = grad.sadvisor
        if (Student_Info_Changed(grad) .EQ. .TRUE.) then
            if (old_advisor .NE. grad.sadvisor) then
##              deleterow #masterfrm studenttbl
            else
##              putrow #masterfrm studenttbl
##                 (sgpa = grad.sgpa,
##                  scomment = grad.scomment,
##                  sadvisor = grad.sadvisor)
        endif
      endif

##    }        /* 'Zoom' */

##    activate menuitem 'Quit', frskey2
##    {
##          breakdisplay
##    }        /* 'Quit' */

##    finalize

##    end      ! Master

!
! Procedure:  Load_Students
! Purpose:    Given an advisor name, load into the 'studenttbl'
!             table field all the students who report to the
!             professor with that name.
! Parameters:
!             advisor - User specified professor name.
!             Uses the global student record.
!

##    subroutine Load_Students(advisor)

##    declare forms
```

```
##    character*(*) advisor

      ! grad student record maps to database table
##    structure /grad_student/
##        character*25    sname
##        integer*2       sage
##        character*25    sbdate
##        real*4          sgpa
##        integer*4       sidno
##        character*200   scomment
##        character*25       sadvisor
##    end structure
##    record /grad_student/ grad

      !
      ! Clear previous contents of table field. Load the table
      ! field from the database table based on the advisor name.
      ! Columns 'sname' and 'sage' will be displayed, and all
      ! others will be hidden.
      !
##    message 'Retrieving Student Information . . .'
##    clear field studenttbl
##    retrieve
##         (grad.sname = s.sname,
##          grad.sage = s.sage,
##          grad.sbdate = s.sbdate,
##          grad.sgpa = s.sgpa,
##          grad.sidno = s.sidno,
##          grad.scomment = s.scomment,
##          grad.sadvisor = s.sadvisor)
##          where s.sadvisor = advisor
##    {
##          loadtable #masterfrm studenttbl
##               (sname = grad.sname,
##                sage = grad.sage,
##                sbdate = grad.sbdate,
##                sgpa = grad.sgpa,
##                sidno = grad.sidno,
##                scomment = grad.scomment,
##                sadvisor = grad.sadvisor)
##    }

##    end    Load_Students

!
! Procedure:  Student_Info_Changed
! Purpose:    Allow the user to zoom into the details of a
!             selected student. Some of the data can be updated
!             by the user. If any updates were made, then reflect
!             these back into the database
!             table. The procedure returns TRUE if
!             any changes were made.
! Parameters:
!             None - Uses with data in the global 'grad' record.
! Returns:
!             TRUE/FALSE - Changes were made to the database.
!             Sets the global 'grad' record with the new data.
!

##    logical function Student_Info_Changed(grad)

##    declare forms
      ! grad student record maps to database table
##    structure /grad_student/
##        character*25 sname
```

```
##          integer*2 sage
##          character*25   sbdate
##          real*4         sgpa
##          integer*4      sidno
##          character*200  scomment
##          character*25   sadvisor
##      end structure
##      record /grad_student/ grad

##      integer*4 changed       ! Changes made to data in form
##      integer*4 valid_advisor ! Valid advisor name ?

        external studentfrm
##      integer*4 studentfrm ! Compiled form

        ! Control ADDFORM to only initialize once
        integer*4 loadform
        data loadform/0/

        if (loadform .EQ. 0) then
##          message 'Loading Student form . . .'
##          addform studentfrm
            loadform = 1
        endif

##      display #studentfrm fill
##      initialize
##              (sname     = grad.sname,
##               sage      = grad.sage,
##               sbdate    = grad.sbdate,
##               sgpa      = grad.sgpa,
##               sidno     = grad.sidno,
##               scomment  = grad.scomment,
##               sadvisor  = grad.sadvisor)

##      activate menuitem 'Write'
##      {
            !
            ! If changes were made then update the database
            ! table. Only bother with the fields that are not
            ! read-only.
            !
##          inquire_frs form (changed = change)

            if (changed .EQ. 1) then
##              validate
##              message 'Writing changes to database. . .'

##              getform
##                 (grad.sgpa = sgpa,
##                  grad.scomment = scomment,
##                  grad.sadvisor = sadvisor)

                    ! Enforce integrity of professor name
                    valid_advisor = 0
##                  retrieve (valid_advisor = 1)
##                      where p.pname = grad.sadvisor

                    if (valid_advisor .EQ. 0) then
##                      message 'Not a valid advisor name'
##                      sleep 2
##                      resume field sadvisor
                    else
##                      replace s (sgpa = grad.sgpa,
##                          scomment = grad.scomment,
##                          sadvisor = grad.sadvisor)
```

```
##                          where s.sidno = grad.sidno
##                      breakdisplay
                    endif
          endif
##    }               /* 'Write' */

##    activate menuitem 'End', frskey3
##    {
              ! Quit without submitting changes
              changed = 0
##            breakdisplay
##    }               /* 'Quit' */

##    finalize

      if ( changed .EQ. 1) then
           Student_Info_Changed = .TRUE.
      else
           Student_Info_Changed = .FALSE.
      endif

      return
##    end
```

UNIX, VMS, Windows—An Interactive Database Browser Using Param Statements

This application lets the user browse and update data in any table in any database. You should already have used VIFRED to create a default form based on the database table to be browsed. VIFRED builds a form whose fields have the same names and data types as the columns of the database table specified.

The program prompts the user for the name of the database, the table, and the form. In the **Get_Form_Data** procedure, it uses the **formdata** statement to find out the name, data type and length of each field on the form. It uses this information to dynamically build the elements for the **param** versions of the **retrieve**, **append**, **putform** and **getform** statements. These elements include the **param** target string, which describes the data to be processed, and the array of variable addresses, which informs the statement where to get or put the data. The type information the **formdata** statement collects includes the option of making a field nullable. If a field is nullable, the program builds a target string that specifies the use of a null indicator, and it sets the corresponding element of the array of variable addresses to point to a null indicator variable.

After the components of the **param** clause are built, the program displays the form. If the user selects the **Browse** menu item, the program uses a **param** version of the **retrieve** statement to obtain the data. For each row, the **putform** and **redisplay** statements exhibit this data to the user. A **submenu** allows the user to get the next row or to stop browsing. When the user selects the **Insert** menu item, the program uses the **param** versions of the **getform** and **append** statements to add a new row to the database.

The application runs in the UNIX, VMS, and Windows environments.

UNIX

```
C
C   Procedure: main
C   Purpose:   Start up program and Ingres, prompting user
C              for names of form and table. Call Get_Form_Data() to
C              obtain profile of form. Then allow user to
C              interactively browse the database table and/or
C              APPEND new data.

##  program main

##  declare forms
C   Global declarations
C
C   Target string buffers for use in PARAM clauses of GETFORM,
C   PUTFORM, APPEND and RETRIEVE statements. Note that the APPEND
C   and PUTFORM statements have the same target string syntax.
C   Therefore in this application, because the form used
C   corresponds exactly to the database table, these two statements
C   can use the same target string, 'putlst'.
C
C                  For APPEND and PUTFORM statements
##     character*1000    putlst
C                  For GETFORM statement
##     character*1000    getlst
C                  For RETRIEVE statement

##     character*1000    rtnlst

       integer MAXCOL, BUFSIZ
C                  DB maximum number of columns
       parameter (MAXCOL = 127)
C                  Size of 'pool' of char strings
       parameter (BUFSIZ = 3000)

C
C   An array of addresses of program data for use in the PARAM
C   clauses. This array will be initialized by the program to
C   point to variables and null indicators.
C
C                Addresses of vars and inds
##         integer*4 varadr(MAXCOL*2)

C
C   Variables for holding data of type integer, float and
C   character string. Note that to economize on memory usage,
C   character data is managed as segments on one large array,
C   'chvars'. Numeric variables and indicators are managed as an
C   array of structures. The addresses of these data areas
C   are assigned to the 'varadr' array, according to the type of
C   the field/database column.
C
C              Pool for character data
            character*(BUFSIZ) chvars

C             For integer data
        integer*4         intv(MAXCOL)
C              For floating-point data
        double precision  fltv(MAXCOL)
C              For null indicators
        integer*2         indv(MAXCOL)

##      character*25 dbname, frmnam, tabnam
C              Catch database and forms errors
##      integer*4          inqerr
```

```
C               Catch error on database APPENDs
##      integer*4          numchg
C               Browse flag
        logical           getnxt
C               Logical function (see below)
        logical           GetFrm

        putlst = ' '
        etlst = ' '
        rtnlst = ' '
        chvars = ' '

##      forms

##      prompt ('Database name: ', dbname)

C       '-E' flag tells Ingres not to quit on
C       start-up errors
##      ingres '-E' dbname
##      inquire_ingres (inqerr = ERRORNO)
         f (inqerr .GT. 0) then
##          message 'Could not start Ingres. Exiting.'
##          endforms
##          exit
            call exit
        endif
C       Prompt for table and form names
##      prompt ('Table name: ', tabnam)
##      range of t IS tabnam
##      inquire_ingres (inqerr = ERRORNO)
         if (inqerr .GT. 0) then
##           message 'Non-existent table. Exiting.'
##           endforms
##           exit
             call exit
        endif

##      prompt ('Form name: ', frmnam)

##      forminit frmnam
C       All forms errors are reported through INQUIRE_FRS
##      inquire_frs FRS (inqerr = ERRORNO)
        if (inqerr .GT. 0) then
##          message 'Could not access form. Exiting.'
##          endforms
##          exit
            call exit
         endif

C
C        Get profile of form. Construct target lists and access
C        variables for use in queries to browse and update data.
C        if (.NOT. GetFrm (frmnam, putlst, getlst, rtnlst, varadr,
    &              chvars, intv, fltv, indv)) then

##            message 'Could not profile form. Exiting.'
##            endforms
##            exit
              call exit
          endif

C
C      Display form and interact with user, allowing browsing and
C      appending of new data.
C
##     display frmnam fill
```

```
##     initialize
##     activate menuitem 'Browse'
##     {
C
C      Retrieve data and display first row on form, allowing user
C      to browse through successive rows. If data types from table
C      are not consistent with data descriptions obtained from
C      user's form, a retrieval error will occur. Inform user of
C      this or other errors.
C      Sort on first column. Note the use of 'ret_varN' to indicate
C      the column name to sort on.
C
##     retrieve (param(rtnlst, varadr))
##            sort by ret_var1
##     {               getnxt = .FALSE.
##            putform frmnam (param(putlst, varadr))

##            inquire_frs frs    (inqerr = ERRORNO)
              if (inqerr .GT. 0) then
##                message 'Could not put data into form'
##                endretrieve
              endif

C             Display data before prompting user with submenu
##            redisplay
##            submenu
##            activate menuitem 'Next', frskey4
##            {
##                  message 'Next row'
                    getnxt = .TRUE.
##            }
##            activate menuitem 'End', frskey3
##            {
##                        endretrieve
##             }
##          }    /* End of RETRIEVE Loop */

##          inquire_ingres (inqerr = ERRORNO)
            if (inqerr .GT. 0) then
##              message 'Could not retrieve data from database'
            else if (getnxt) then
C                Retrieve loop ended because of no more rows
##               message 'No more rows'
            endif

##          sleep 2

C           Clear fields filled in submenu operations
##          clear field all
##     }

##     activate menuitem 'Insert'
##     {

##          getform frmnam (param(getlst, varadr))
##          inquire_frs frs (inqerr = ERRORNO)
            if (inqerr .GT. 0) then
##               clear field all
##               resume
            endif

##          append to tabnam (param(putlst, varadr))

##          inquire_ingres (inqerr = ERRORNO, numchg = ROWCOUNT)
      if ((inqerr .GT. 0) .OR. (numchg .EQ. 0)) then
##               message 'No rows appended because of error.'
```

```
       else
##                message 'One row inserted'
       endif
##     sleep 2
##     }

##     activate menuitem 'Clear'
##     {
##            clear field all
##     }

##     activate menuitem 'End', frskey3
##     {
##            breakdisplay
##     }

##     finalize
##     endforms
##     exit
##     end

C
C Procedure: GetFrm
C Purpose:   Get the name and data type of each field of a form
C            using the FORMDATA loop. From this information, build
C            the target strings and array of variable addresses
C            for use in the PARAM target list of database an
C            and forms statements. For example, assume the
C            form has the following fields:
C
C             Field name      Type          Nullable?
C             ----------      ----          -------
C             name            character     No
C             age             integer       Yes
C             salary          money         Yes
C
C            Based on this form, this procedure will construct the
C            following target string for the PARAM clause of a
C            PUTFORM statement:
C
C             'name = %c, age = %i4:%i2, salary = %f8:i2'
C
C            Note that the target strings for other statements have
C            differing syntax, depending on whether the
C            field/columnname or the user variable is the target of
C            the statement.
C
C            The other element of the PARAM clause, the 'varadr'
C            array, would be constructed by this procedure as
C            follows:
C
C              varadr(1) = pointer into 'chvars' array
C              varadr(2) = address of intv(1)
C              varadr(3) = address of indv(1)
C              varadr(4) = address of fltv(2)
C              varadr(5) = address of indv(2)
C
C
##     logical function GetFrm (frmnam, putlst, getlst, rtnlst,
##    &                         varadr, chvars, intv, fltv, indv)

##     declare forms
##     character*(*)   frmnam

C               For APPEND and PUTFORM statements
##     character*(*)   putlst
```

```
C               For GETFORM statement
##     character*(*)   getlst
C               For RETRIEVE statement
##     character*(*)   rtnlst

C              DB maximum number of columns
       integer*4      MAXCOL
       parameter     (MAXCOL = 127)

C              Addresses of vars and inds

        integer*4      varadr(MAXCOL*2)
C               Pool for character data
        character*(*)  chvars

C               For integer data
        integer*4           intv(*)
C               For floating-point data
        double precision    fltv(*)
C               For null indicators
        integer*2           indv(*)

##      integer*4           inqerr
C               Data type of field
##      integer*4           fldtyp
C               Name of field
##      character*25        fldnam
C               Length of field name
        integer*4           fldlen
C               Size of (character) field
##      integer*4           fldsiz
C               Is field a table field?
##      integer*4           istbl
C               Index into variable address array
        integer*4           numadr
C               Current field number
        integer*4           fldcnt
C               Return status
        logical             rtnsts
C               Length of character buffer
        integer*4           chvlen

C       following 4 variables tell where to assign next character
C               Index into putlst
        integer*4           putcnt
C               Index into getlst
        integer*4           getcnt
C               Index into rtnlst
        integer*4           rtncnt
C               Index into character pool
        integer*4           chrptr

C       Data types of fields on form
        integer*2 DATE, MONEY, CHAR, VARCHAR, INT, FLOAT, C, TEXT
      parameter   (DATE       =        3,
      &            MONEY      =        5,
      &            CHAR       =        20,
      &            VARCHAR    =        21,
      &            INT        =        30,
      &            FLOAT      =        31,
      &            C          =        32,
      &            TEXT       =        37 )

        rtnsts = .TRUE.
        numadr = 1
```

```
        putcnt = 1
        getcnt = 1
        rtncnt = 1
        chrptr = 1
        fldcnt = 1

        chvlen = len(chvars)

##      formdata frmnam
##      {
C           Get data information and name of each field
##          inquire_frs field '' (fldtyp = DATATYPE, fldnam = NAME,
##                   fldsiz = LENGTH, istbl = TABLE)

C           Return on errors
##          inquire_frs frs (inqerr = ERRORNO)
            if (inqerr .GT. 0) then
               rtnsts = .FALSE.
##             enddata
            endif

C
C          This application does not process table fields. However,
C          the TABLEDATA statement is available to profile table
C          fields.
C           if (istbl .EQ. 1) then
##              message 'Table field in form'
##              sleep 2
                rtnsts = .FALSE.
##              enddata
            endif

C           More fields than allowable columns in database?
            if (fldcnt .GT. MAXCOL) then
##             message 'Number of fields exceeds allowable
##                      database columns'
##             sleep 2
               rtnsts = .FALSE.
##             enddata
               endif

C            Separate target list items with commas
               if (fldcnt .GT. 1) then
                    putlst(putcnt:) = ','
                    putcnt = putcnt + 1

                    getlst(getcnt:) = ','
                    getcnt = getcnt + 1

                    rtnlst(rtncnt:) = ','
                    rtncnt = rtncnt + 1
               endif

C            Calculate the length of fldnam without trailing spaces
             fldlen = len(fldnam)
1000         continue
             if ((fldlen .GT. 1) .AND.
                  (fldnam(fldlen:fldlen) .EQ. ' ')) then
                    fldlen = fldlen - 1
                    goto 1000
             end if

C            Field/column name is the target in
C            PUTFORM/APPEND statements
             putlst(putcnt:) = fldnam
             putcnt = putcnt + fldlen
```

```
C
C            Enter data type information in target list. Point
C            array of addresses into relevant data pool.
C            Note that bytesting the absolute
C            value of the data type value, the
C            program defers the question of nullable data to a
C            later segment of the code, where it is handled in
C            common for all types. (Recall that a negative data
C            type indicates a nullable field.)
C
               if (abs(fldtyp) .EQ. INT) then
                  putlst(putcnt:) = '=%i4'
                  putcnt = putcnt + 4

                  getlst(getcnt:) = '%i4'
                  getcnt = getcnt + 3

                  rtnlst(rtncnt:) = '%i4'
                  rtncnt = rtncnt + 3

                  varadr(numadr) = IInum(intv(fldcnt))
                  numadr = numadr + 1

            else if ( (abs(fldtyp) .EQ. FLOAT) .OR.
     &                (abs(fldtyp) .EQ. MONEY) ) then

                  putlst(putcnt:) = '=%f8'
                  putcnt = putcnt + 4

                  getlst(getcnt:) = '%f8'
                  getcnt = getcnt + 3

                  rtnlst(rtncnt:) = '%f8'
                  rtncnt = rtncnt + 3

                  varadr(numadr) = IInum(fltv(fldcnt))
                  numadr = numadr + 1

            else if ((abs(fldtyp) .EQ. C)            .OR.
     &              (abs(fldtyp)  .EQ. CHAR)        .OR  .
     &              (abs(fldtyp)  .EQ. TEXT)        .OR.
     &              (abs(fldtyp)  .EQ. VARCHAR)     .OR.
     &              (abs(fldtyp)  .EQ. DATE))      then

                  putlst(putcnt:) = '=%c'
                  putcnt = putcnt + 3

                  getlst(getcnt:) = '%c'
                  getcnt = getcnt + 2

                  rtnlst(rtncnt:) = '%c'
                  rtncnt = rtncnt + 2

C
C                 Assign a segment of character buffer as space for
C                 data associated with this field. If assignment
C                 would cause overflow, give error and return.
C
                  if ( (chrptr + fldsiz) .GT. (chvlen) ) then
##                  message 'Character data fields will
##                  cause overflow'
##                  sleep 2
                    rtnsts = .FALSE.
##                  enddata
                  endif
```

```
                    varadr(numadr) =
                        IIstr(chvars(chrptr:chrptr+fldsiz-1))
                    numadr = numadr + 1
                    chrptr = chrptr + fldsiz

                else
##                message 'Field has unknown data type'
                  rtnsts = .FALSE.
##                enddata
                endif

C
C               If field is nullable, complete target lists and
C               address assignments to allow for null data.
C
                if (fldtyp .LT. 0) then

                     putlst(putcnt:) = ':%i2'
                     putcnt = putcnt + 4

                     getlst(getcnt:) = ':%i2'
                     getcnt = getcnt + 4

                     rtnlst(rtncnt:) = ':%i2'
                     rtncnt = rtncnt + 4

                     varadr(numadr) = IInum(indv(fldcnt))
                     numadr = numadr + 1

                endif

C               Ready for next field
                fldcnt = fldcnt + 1

C               Field/column name is the object in
C               GETFORM/RETRIEVE statements

                getlst(getcnt:) = '='
                getcnt = getcnt + 1
                getlst(getcnt:) = fldnam
                getcnt = getcnt + fldlen

                rtnlst(rtncnt:) = '=t.'
                rtncnt = rtncnt + 3
                rtnlst(rtncnt:) = fldnam
                rtncnt = rtncnt + fldlen

##      }             /* End of FORMDATA loop */

        GetFrm = rtnsts
        return
##      end
```

VMS

```
!
! Procedure: main
! Purpose:   Start up program and Ingres, prompting user
!            for names of form and table. Call Get_Form_Data() to
!            obtain profile of form. Then allow user to
!            interactively browse the database table
!            and/or APPEND new data.
!

##    program main
```

```
##    declare forms

!     Global declarations
!
!   Target string buffers for use in PARAM clauses of GETFORM,
!   PUTFORM, APPEND and RETRIEVE statements. Note that the APPEND
!   and PUTFORM statements have the same target string syntax.
!   Therefore in this application, because the form used
!   corresponds exactly to the database table, these two s
!   statements can use the same target string, 'put_target_list'.
!

##  character*1000 put_target_list
!                               For APPEND and PUTFORM statements
##  character*1000 get_target_list  ! For GETFORM statement
##  character*1000 ret_target_list  ! For RETRIEVE statement

    integer maxcols, charbufsize
    parameter (maxcols = 127)       ! DB maximum number of columns
    parameter (charbufsize = 3000)
!                         Size of 'pool' of char strings

!
!   An array of addresses of program data for use in the PARAM
!   clauses. This array will be initialized by the program to
!   point to variables and null indicators.
!

##      integer*4 var_addresses(MAXCOLS*2)
!                                    Addresses of vars and inds

!
!   Variables for holding data of type integer, float and
!   character string. Note that to economize on memory usage,
!   character data is managed as segments on one large array,
!   'char_vars'. Numeric variables and indicators are managed as
!   an array of structures. The addresses of these data areas
!   are assigned to the 'var_addresses' array, according to the
!   type of the field/database column.
!

      character*(CHARBUFSIZE) char_vars ! Pool for character data

        structure /n_vars/
             integer*4 intv           ! For integer data
             double precision fltv    ! For floating-point data
             integer*2 indv           ! For null indicators
        end structure
        record /n_vars/ vars(MAXCOLS)

##      character*25  dbname, formname, tabname
##      integer*4     inq_error ! Catch database and forms errors
##      integer*4    num_updates ! Catch error on database APPENDs
        logical      want_next      ! Browse flag
        logical      Get_Form_Data  ! Logical function (see below)
        put_target_list = ' '
        get_target_list = ' '
        ret_target_list = ' '
        char_vars = ' '

##      forms

##      prompt ('Database name: ', dbname)

   ! '-E' flag tells Ingres not to quit on start-up errors
```

```
##      ingres '-E' dbname
##      inquire_ingres (inq_error = ERRORNO)
        if (inq_error .GT. 0) then
##          message 'Could not start Ingres. Exiting.'
##          endforms
##          exit
            call exit
        endif

        ! Prompt for table and form names
##      prompt ('Table name: ', tabname)
##      range of t IS tabname
##      inquire_ingres (inq_error = ERRORNO)
        if (inq_error .GT. 0) then
##          message 'Non-existent table. Exiting.'
##          endforms
##          exit
            call exit
        endif

##      prompt ('Form name: ', formname)
##      forminit formname

        ! All forms errors are reported through INQUIRE_FRS
##      inquire_frs frs (inq_error = ERRORNO)
        if (inq_error .GT. 0) then
##          message 'Could not access form. Exiting.'
##          endforms
##          exit
            call exit
        endif

        !
        ! Get profile of form. Construct target lists and access
        ! variables for use in queries to browse and update data.
        !   if (.NOT. Get_Form_Data (formname, put_target_list,
  1               get_target_list, ret_target_list, var_addresses,
  2               char_vars, vars)) then

##          message 'Could not profile form. Exiting.'
##          endforms
##          exit
            call exit
      endif

      !
      ! Display form and interact with user, allowing browsing
      ! and appending of new data.
      !
##    display formname fill
##    initialize
##    activate menuitem 'Browse'
##    {
        !
        ! Retrieve data and display first row on form, allowing
        ! user to browse through successive rows. If data types
        ! from table are not consistent with data descriptions
        ! obtained from user's form, a retrieval error will
        ! occur. Inform user of this or other errors.
        ! Sort on first column. Note the use of 'ret_varN' to
        ! indicate the column name to sort on.
        !

##      retrieve (param(ret_target_list, var_addresses))
##          sort by ret_var1
##      {
```

```
          want_next = .FALSE.
##        putform formname (param(put_target_list, var_addresses))

##          inquire_frs frs (inq_error = errorno)
            if (inq_error .GT. 0) then
##              message 'Could not put data into form'
##              endretrieve
            endif

            ! Display data before prompting user with submenu
##          redisplay
##          submenu
##          activate menuitem 'Next', frskey4
##          {
##              message 'Next row'
                want_next = .TRUE.
##          }
##          activate menuitem 'End', frskey3
##          {
##              endretrieve
##          }
##      }     /* End of RETRIEVE Loop */

##      inquire_ingres (inq_error = errorno)
        if (inq_error .GT. 0) then
##          message 'Could not retrieve data from database'

        else if (want_next) then
            ! Retrieve loop ended because of no more rows
##          message 'No more rows'
        endif

##      sleep 2

        ! Clear fields filled in submenu operations
##      clear field all
##    }

##    activate menuitem 'Insert'
##    {

##        getform formname (param(get_target_list, var_addresses))
##        inquire_frs frs (inq_error = errorno)
          if (inq_error .GT. 0) then
##            clear field all
##            resume
          endif

##        append to tabname (param(put_target_list,
##                var_addresses))

##        inquire_ingres (inq_error = errorno,
##                num_updates = rowcount)
          if ((inq_error .GT. 0) .OR. (num_updates .EQ. 0)) then
##            message 'No rows appended because of error.'
          else
##            message 'One row inserted'
          endif
##        sleep 2
##    }

##    activate menuitem 'Clear'
##    {
##        clear field all
##    }
```

```
##    activate menuitem 'End', frskey3
##    {
##         breakdisplay
##    }

##    finalize
##    endforms
##    exit
##    end

!
! Procedure:   Get_Form_Data
! Purpose:     Get the name and data type of each field of a form
!              using the FORMDATA loop. From this information,
!              build the target strings and array of variable
!              addresses for use in the PARAM target list of
!              database and forms statements.
!              For example, assume the form has the
!              following fields:
!
!
!                 Field name        Type             Nullable?
!                 -------           ------           --------
!                 name              character        No
!                 age               integer          Yes
!                 salary            money            Yes
!
!              Based on this form, this procedure will construct
!              the following target string for the PARAM clause
!              of a PUTFORM statement:
!
!              'name = %c, age = %i4:%i2, salary = %f8:i2'
!
!              Note that the target strings for other statements
!              have differing syntax, depending on whether the
!              field/column name or the user variable is the
!              target of the statement.
!
!              The other element of the PARAM clause, the
!              'var_addresses' array, would be constructed by this
!              procedure as follows:
!
!                  var_addresses(1) =
!                              pointer into 'char_vars' array
!                  var_addresses(2) = address of vars(1).intv
!                  var_addresses(3) = address of vars(1).indv
!                  var_addresses(4) = address of vars(2).fltv
!                  var_addresses(5) = address of vars(2).indv
!
!

#          logical function Get_Form_Data (formname,
    1          put_target_list, get_target_list, ret_target_list,
    2          var_addresses, char_vars, vars)

##         declare forms
##         character*(*) formname

           character*(*) put_target_list
                        ! For APPEND and PUTFORM statements
           character*(*) get_target_list ! For GETFORM statement
           character*(*) ret_target_list ! For RETRIEVE statement

           integer*4 maxcols
           parameter (maxcols = 127)
!                               DB maximum number of columns
```

```
           integer*4 var_addresses(MAXCOLS*2)
                                       ! Addresses of vars and inds
           character*(*) char_vars    ! Pool for character data

      structure /n_vars/
           integer*4 intv              ! For integer data
           double precision fltv       ! For floating-point data
           integer*2 indv              ! For null indicators
      end structure
      record /n_vars/ vars(MAXCOLS)

##        integer*4    inq_error
##        integer*4    fld_type      ! Data type of field
##        character*25 fld_name      ! Name of field
          integer*4    fld_name_len  ! Length of field name
##        integer*4    fld_length    ! Length of (character) field
##        integer*4    is_table      ! Is field a table field?
          character*15 loc_target    ! Temporary target description
          integer*4    addr_cnt ! Index into variable address array
          integer*4    fld_cnt       ! Current field number
          logical      ret_stat      ! Return status
          integer*4    char_vars_len ! Length of character buffer
          ! following 4 variables tell where to assign next
          ! character
          integer*4    put_cnt       ! Index into put_target_list
          integer*4    get_cnt       ! Index into get_target_list
          integer*4    ret_cnt       ! Index into ret_target_list
          integer*4    char_ptr     ! Index into character pool

          ! Data types of fields on form
        integer*2 date, money, char, varchar, int, float, c, text
          parameter     (date          = 3,
    1                    money          = 5,
    2                    char           = 20,
    3                    varchar        = 21,
    4                    int            = 30,
    5                    float          = 31,
    6                    c              = 32,
    7                    text           = 37 )

          ret_stat   = .TRUE.
          addr_cnt   = 1

          put_cnt    = 1
          get_cnt    = 1
          ret_cnt    = 1
          char_ptr   = 1
          fld_cnt    = 1
          char_vars_len = len(char_vars)

##      formdata formname
##      {
          ! Get data information and name of each field
##    inquire_frs field '' (fld_type = datatype, fld_name = name,
##                          fld_length = length, is_table = table)

          ! Return on errors
##        inquire_frs frs (inq_error = errorno)
          if (inq_error .gt. 0) then
              ret_stat = .false.
##            enddata
          endif

          !
          ! This application does not process table fields.
```

```
          ! However, the TABLEDATA statement is available to
          ! profile table fields.
          !
          if (is_table .EQ. 1) then
##            message 'Table field in form'
##            sleep 2
              ret_stat = .FALSE.
##            enddata
          endif

          ! More fields than allowable columns in database?
          if (fld_cnt .GT. MAXCOLS) then
##            message
##              'Number of fields exceeds allowable database
##               columns'
##            sleep 2
              ret_stat = .FALSE.
##            enddata
          endif

          ! Separate target list items with commas
          if (fld_cnt .GT. 1) then
              put_target_list(put_cnt:) = ','
              put_cnt = put_cnt + 1

              get_target_list(get_cnt:) = ','
              get_cnt = get_cnt + 1

              ret_target_list(ret_cnt:) = ','
              ret_cnt = ret_cnt + 1
          endif

          ! Calculate the length of fld_name without trailing
          ! spaces
          fld_name_len = len(fld_name)
          do while ((fld_name_len .GT. 1) .AND.
     1              (fld_name(fld_name_len:fld_name_len) .EQ. ' '))
              fld_name_len = fld_name_len - 1
          end do

          ! Field/column name is the target in PUTFORM/APPEND
          ! statements
          put_target_list(put_cnt:) = fld_name
          put_cnt = put_cnt + fld_name_len

          !
          ! Enter data type information in target list. Point
          ! array of addresses into relevant data pool. Note that
          ! by testing the absolute value of the data type value,
          ! the program defers the question of nullable data to a
          ! later segment of the code, where it is handled in
          ! common for all types. (Recall that a negative data
          ! type indicates a nullable field.)
          !
          if (abs(fld_type) .EQ. INT) then
              put_target_list(put_cnt:) = '=%i4'
              put_cnt = put_cnt + 4

      get_target_list(get_cnt:) = '%i4'
      get_cnt = get_cnt + 3

      ret_target_list(ret_cnt:) = '%i4'
      ret_cnt = ret_cnt + 3

      var_addresses(addr_cnt) = %loc(vars(fld_cnt).intv)
      addr_cnt = addr_cnt + 1
```

```
    else if ( (abs(fld_type) .eq. float) .or.
   1  (abs(fld_type) .eq. money) ) then

    put_target_list(put_cnt:) = '=%f8'
    put_cnt = put_cnt + 4

    get_target_list(get_cnt:) = '%f8'
    get_cnt = get_cnt + 3

    ret_target_list(ret_cnt:) = '%f8'
    ret_cnt = ret_cnt + 3

    var_addresses(addr_cnt) = %loc(vars(fld_cnt).fltv)
    addr_cnt = addr_cnt + 1

    else if ( (abs(fld_type)   .eq. c)                .or.
   1  (abs(fld_type)  .eq. char)             .or.
   2  (abs(fld_type)  .eq. text)             .or.
   3  (abs(fld_type)  .eq. varchar)          .or.
   4  (abs(fld_type)  .eq. date) )           then

    put_target_list(put_cnt:) = '=%c'
    put_cnt = put_cnt + 3

    get_target_list(get_cnt:) = '%c'
    get_cnt = get_cnt + 2

    ret_target_list(ret_cnt:) = '%c'
    ret_cnt = ret_cnt + 2

    !
    ! Assign a segment of character buffer as space for
    ! data associated with this field. If assignment would
    ! cause overflow, give error and return.
    !

    if ( (char_ptr + fld_length) .gt.
    1                          (char_vars_len) ) then
##  message 'Character data fields will cause overflow'
##  sleep 2
    ret_stat = .FALSE.
##  enddata
    endif

    var_addresses(addr_cnt) =
    1  IIdesc(char_vars(char_ptr:char_ptr+fld_length-1))
    addr_cnt = addr_cnt + 1
    char_ptr = char_ptr + fld_length

          else
##            message 'Field has unknown data type'
              ret_stat = .false.
##            enddata

          endif
          !
          ! If field is nullable, complete target lists and
          ! address assignments to allow for null data.
          !
          if (fld_type .LT. 0) then

              put_target_list(put_cnt:) = ':%i2'
              put_cnt = put_cnt + 4

              get_target_list(get_cnt:) = ':%i2'
```

```
                get_cnt = get_cnt + 4

                ret_target_list(ret_cnt:) = ':%i2'
                ret_cnt = ret_cnt + 4

                var_addresses(addr_cnt) = %loc(vars(fld_cnt).indv)
                addr_cnt = addr_cnt + 1

              endif

            ! Ready for next field
            fld_cnt = fld_cnt + 1

            ! Field/column name is the object in
            ! getform/retrieve statements

              get_target_list(get_cnt:) = '='
              get_cnt = get_cnt + 1
              get_target_list(get_cnt:) = fld_name
              get_cnt = get_cnt + fld_name_len

              ret_target_list(ret_cnt:) = '=t.'
              ret_cnt = ret_cnt + 3
              ret_target_list(ret_cnt:) = fld_name
              ret_cnt = ret_cnt + fld_name_len

##          }            /* End of FORMDATA loop */
            Get_Form_Data = ret_stat
            return
##          end
```

Windows

```
!
! Procedure: main
! Purpose:   Start up program and Ingres, prompting user
!            for names of form and table. Call Get_Form_Data() to
!            obtain profile of form. Then allow user to
!            interactively browse the database table
!            and/or APPEND new data.
!

##    program main

##    declare forms

!     Global declarations
!
!   Target string buffers for use in PARAM clauses of GETFORM,
!   PUTFORM, APPEND and RETRIEVE statements. Note that the APPEND
!   and PUTFORM statements have the same target string syntax.
!   Therefore in this application, because the form used
!   corresponds exactly to the database table, these two s
!   statements can use the same target string, 'put_target_list'.
!

##  character*1000 put_target_list
!                               For APPEND and PUTFORM statements
##  character*1000 get_target_list  ! For GETFORM statement
##  character*1000 ret_target_list  ! For RETRIEVE statement

        integer MAXCOLS, CHARBUFSIZE
        parameter (MAXCOLS = 127)       ! DB maximum number of columns
        parameter (CHARBUFSIZE = 3000)
```

```
!                        Size of 'pool' of char strings

!
!   An array of addresses of program data for use in the PARAM
!   clauses. This array will be initialized by the program to
!   point to variables and null indicators.
!

##      integer*4 var_addresses(MAXCOLS*2)
!                                   Addresses of vars and inds

!
!   Variables for holding data of type integer, float and
!   character string. Note that to economize on memory usage,
!   character data is managed as segments on one large array,
!   'char_vars'. Numeric variables and indicators are managed as
!   an array of structures. The addresses of these data areas
!   are assigned to the 'var_addresses' array, according to the
!   type of the field/database column.
!

      character*(CHARBUFSIZE) char_vars ! Pool for character data

        structure /n_vars/
             integer*4 intv           ! For integer data
             double precision fltv    ! For floating point data
             integer*2 indv           ! For null indicators
        end structure
        record /n_vars/ vars(MAXCOLS)

##      character*25  dbname, formname, tabname
##      integer*4     inq_error ! Catch database and forms errors
##      integer*4    num_updates ! Catch error on database APPENDs
        logical      want_next      ! Browse flag
        logical      Get_Form_Data  ! Logical function (see below)
        put_target_list = ' '
        get_target_list = ' '
        ret_target_list = ' '
        char_vars = ' '

##      forms

##      prompt ('Database name: ', dbname)

        ! '-E' flag tells Ingres not to quit on start-up errors
##      ingres '-E' dbname
##      inquire_ingres (inq_error = ERRORNO)
        if (inq_error .GT. 0) then
##          message 'Could not start Ingres. Exiting.'
##          endforms
##          exit
            call exit
        endif

        ! Prompt for table and form names
##      prompt ('Table name: ', tabname)
##      range of t IS tabname
##      inquire_ingres (inq_error = ERRORNO)
        if (inq_error .GT. 0) then
##          message 'Non-existent table. Exiting.'
##          endforms
##          exit
            call exit
        endif

##      prompt ('Form name: ', formname)
```

```
##      forminit formname

        ! All forms errors are reported through INQUIRE_FRS
##      inquire_frs frs (inq_error = ERRORNO)
        if (inq_error .GT. 0) then
##          message 'Could not access form. Exiting.'
##          endforms
##          exit
            call exit
        endif

        !
        ! Get profile of form. Construct target lists and access
        ! variables for use in queries to browse and update data.
        if (.NOT. Get_Form_Data (formname, put_target_list,
     1              get_target_list, ret_target_list, var_addresses,
     2              char_vars, vars)) then

##          message 'Could not profile form. Exiting.'
##          endforms
##          exit
            call exit
      endif

      !
      ! Display form and interact with user, allowing browsing
      ! and appending of new data.
      !
##    display formname fill
##    initialize
##    activate menuitem 'Browse'
##    {
        !
        ! Retrieve data and display first row on form, allowing
        ! user to browse through successive rows. If data types
        ! from table are not consistent with data descriptions
        ! obtained from user's form, a retrieval error will
        ! occur. Inform user of this or other errors.
        ! Sort on first column. Note the use of 'ret_varN' to
        ! indicate the column name to sort on.
        !

##      retrieve (param(ret_target_list, var_addresses))
##          sort by ret_var1
##      {
          want_next = .FALSE.
##        putform formname (param(put_target_list, var_addresses))

##          inquire_frs frs (inq_error = errorno)
            if (inq_error .GT. 0) then
##              message 'Could not put data into form'
##              endretrieve
            endif

            ! Display data before prompting user with submenu
##          redisplay
##          submenu
##          activate menuitem 'Next', frskey4
##          {
##              message 'Next row'
                want_next = .TRUE.
##          }
##          activate menuitem 'End', frskey3
##          {
##              endretrieve
##          }
```

```
##      }     /* End of RETRIEVE Loop */

##      inquire_ingres (inq_error = errorno)
        if (inq_error .GT. 0) then
##          message 'Could not retrieve data from database'

        else if (want_next) then
            ! Retrieve loop ended because of no more rows
##          message 'No more rows'
        endif

##      sleep 2

        ! Clear fields filled in submenu operations
##      clear field all
##    }

##    activate menuitem 'Insert'
##    {

##        getform formname (param(get_target_list, var_addresses))
##        inquire_frs frs (inq_error = errorno)
          if (inq_error .GT. 0) then
##            clear field all
##            resume
          endif

##        append to tabname (param(put_target_list,
##                var_addresses))

##        inquire_ingres (inq_error = errorno,
##                num_updates = rowcount)
          if ((inq_error .GT. 0) .OR. (num_updates .EQ. 0)) then
##            message 'No rows appended because of error.'
          else
##            message 'One row inserted'
          endif
##        sleep 2
##    }

##    activate menuitem 'Clear'
##    {
##        clear field all
##    }

##    activate menuitem 'End', frskey3
##    {
##         breakdisplay
##    }

##    finalize
##    endforms
##    exit
##    end

!
! Procedure:   Get_Form_Data
! Purpose:     Get the name and data type of each field of a form
!              using the FORMDATA loop. From this information,
!              build the target strings and array of variable
!              addresses for use in the PARAM target list of
!              database and forms statements.
!              For example, assume the form has the
!              following fields:
!
!
```

```
!               Field name        Type            Nullable?
!               -------           ------          --------
!               name              character       No
!               age               integer         Yes
!               salary            money           Yes
!
!             Based on this form, this procedure will construct
!             the following target string for the PARAM clause
!             of a PUTFORM statement:
!
!             'name = %c, age = %i4:%i2, salary = %f8:i2'
!
!             Note that the target strings for other statements
!             have differing syntax, depending on whether the
!             field/column name or the user variable is the
!             target of the statement.
!
!             The other element of the PARAM clause, the
!             'var_addresses' array, would be constructed by this
!             procedure as follows:
!
!                 var_addresses(1) =
!                             pointer into 'char_vars' array
!                 var_addresses(2) = address of vars(1).intv
!                 var_addresses(3) = address of vars(1).indv
!                 var_addresses(4) = address of vars(2).fltv
!                 var_addresses(5) = address of vars(2).indv
!
!

##          logical function Get_Form_Data (formname,
     1          put_target_list, get_target_list, ret_target_list,
     2          var_addresses, char_vars, vars)

##         declare forms
##         character*(*) formname

           character*(*) put_target_list
                        ! For APPEND and PUTFORM statements
           character*(*) get_target_list ! For GETFORM statement
           character*(*) ret_target_list ! For RETRIEVE statement

           integer*4 MAXCOLS
           parameter (MAXCOLS = 127)
!                               DB maximum number of columns

           integer*4 var_addresses(MAXCOLS*2)
                                    ! Addresses of vars and inds
           character*(*) char_vars    ! Pool for character data

      structure /n_vars/
           integer*4 intv              ! For integer data
           double precision fltv       ! For floating point data
           integer*2 indv              ! For null indicators
      end structure
      record /n_vars/ vars(MAXCOLS)

##         integer*4    inq_error
##         integer*4    fld_type      ! Data type of field
##         character*25 fld_name      ! Name of field
           integer*4    fld_name_len  ! Length of field name
##         integer*4    fld_length    ! Length of (character) field
##         integer*4    is_table      ! Is field a table field?
           character*15 loc_target    ! Temporary target description
           integer*4    addr_cnt ! Index into variable address array
           integer*4    fld_cnt       ! Current field number
```

```
          logical      ret_stat      ! Return status
          integer*4    char_vars_len ! Length of character buffer
          ! following 4 variables tell where to assign next
          ! character
          integer*4    put_cnt       ! Index into put_target_list
          integer*4    get_cnt       ! Index into get_target_list
          integer*4    ret_cnt       ! Index into ret_target_list
          integer*4    char_ptr     ! Index into character pool

          ! Data types of fields on form
        integer*2 date, money, char, varchar, int, float, c, text
          parameter     (date          = 3,
     1                   money          = 5,
     2                   char           = 20,
     3                   varchar        = 21,
     4                   int            = 30,
     5                   float          = 31,
     6                   c              = 32,
     7                   text           = 37 )

          ret_stat   = .TRUE.
          addr_cnt   = 1

          put_cnt    = 1
          get_cnt    = 1
          ret_cnt    = 1
          char_ptr   = 1
          fld_cnt    = 1
          char_vars_len = LEN(char_vars)

##      formdata formname
##      {
          ! Get data information and name of each field
##    inquire_frs field '' (fld_type = datatype, fld_name = name,
##                          fld_length = length, is_table = table)

          ! Return on errors
##        inquire_frs frs (inq_error = errorno)
          if (inq_error .gt. 0) then
              ret_stat = .false.
##            enddata
          endif

          !
          ! This application does not process table fields.
          ! However, the TABLEDATA statement is available to
          ! profile table fields.
          !
          if (is_table .EQ. 1) then
##            message 'Table field in form'
##            sleep 2
              ret_stat = .FALSE.
##            enddata
          endif

          ! More fields than allowable columns in database?
          if (fld_cnt .GT. MAXCOLS) then
##            message
##              'Number of fields exceeds allowable database columns'
##            sleep 2
              ret_stat = .FALSE.
##            enddata
          endif

          ! Separate target list items with commas
          if (fld_cnt .GT. 1) then
```

```
            put_target_list(put_cnt:) = ','
            put_cnt = put_cnt + 1

            get_target_list(get_cnt:) = ','
            get_cnt = get_cnt + 1

            ret_target_list(ret_cnt:) = ','
            ret_cnt = ret_cnt + 1
        endif

        ! Calculate the length of fld_name without trailing
        ! spaces
        fld_name_len = LEN(fld_name)
        do while ((fld_name_len .GT. 1) .AND.
1                 (fld_name(fld_name_len:fld_name_len) .EQ. ' '))
            fld_name_len = fld_name_len - 1
        end do

        ! Field/column name is the target in PUTFORM/APPEND
        ! statements
        put_target_list(put_cnt:) = fld_name
        put_cnt = put_cnt + fld_name_len

        !
        ! Enter data type information in target list. Point
        ! array of addresses into relevant data pool. Note that
        ! by testing the absolute value of the data type value,
        ! the program defers the question of nullable data to a
        ! later segment of the code, where it is handled in
        ! common for all types. (Recall that a negative data
        ! type indicates a nullable field.)
        !
        if (abs(fld_type) .EQ. INT) then
            put_target_list(put_cnt:) = '=%i4'
            put_cnt = put_cnt + 4

       get_target_list(get_cnt:) = '%i4'
       get_cnt = get_cnt + 3

       ret_target_list(ret_cnt:) = '%i4'
       ret_cnt = ret_cnt + 3

       var_addresses(addr_cnt) = %loc(vars(fld_cnt).intv)
       addr_cnt = addr_cnt + 1

       else if ( (abs(fld_type) .eq. float) .or.
1      (abs(fld_type) .eq. money) ) then

       put_target_list(put_cnt:) = '=%f8'
       put_cnt = put_cnt + 4

       get_target_list(get_cnt:) = '%f8'
       get_cnt = get_cnt + 3

       ret_target_list(ret_cnt:) = '%f8'
       ret_cnt = ret_cnt + 3

       var_addresses(addr_cnt) = %loc(vars(fld_cnt).fltv)
       addr_cnt = addr_cnt + 1

       else if ( (abs(fld_type)   .eq. c)                .or.
1      (abs(fld_type)  .eq. char)             .or.
2      (abs(fld_type)  .eq. text)             .or.
3      (abs(fld_type)  .eq. varchar)          .or.
4      (abs(fld_type)  .eq. date) )           then
```

```
        put_target_list(put_cnt:) = '=%c'
        put_cnt = put_cnt + 3

        get_target_list(get_cnt:) = '%c'
        get_cnt = get_cnt + 2

        ret_target_list(ret_cnt:) = '%c'
        ret_cnt = ret_cnt + 2

    !
    ! Assign a segment of character buffer as space for
    ! data associated with this field. If assignment would
    ! cause overflow, give error and return.
    !

        if ( (char_ptr + fld_length) .gt.
     1                          (char_vars_len) ) then
##  message 'Character data fields will cause overflow'
##  sleep 2
        ret_stat = .FALSE.
##  enddata
        endif

        var_addresses(addr_cnt) =
     1  IIdesc(char_vars(char_ptr:char_ptr+fld_length-1))
        addr_cnt = addr_cnt + 1
        char_ptr = char_ptr + fld_length

          else
##            message 'Field has unknown data type'
              ret_stat = .false.
##            enddata

          endif
          !
          ! If field is nullable, complete target lists and
          ! address assignments to allow for null data.
          !
          if (fld_type .LT. 0) then

              put_target_list(put_cnt:) = ':%i2'
              put_cnt = put_cnt + 4

              get_target_list(get_cnt:) = ':%i2'
              get_cnt = get_cnt + 4

              ret_target_list(ret_cnt:) = ':%i2'
              ret_cnt = ret_cnt + 4

              var_addresses(addr_cnt) = %loc(vars(fld_cnt).indv)
              addr_cnt = addr_cnt + 1

          endif

        ! Ready for next field
        fld_cnt = fld_cnt + 1

        ! Field/column name is the object in
        ! getform/retrieve statements

          get_target_list(get_cnt:) = '='
          get_cnt = get_cnt + 1
          get_target_list(get_cnt:) = fld_name
          get_cnt = get_cnt + fld_name_len

          ret_target_list(ret_cnt:) = '=t.'
```

```
          ret_cnt = ret_cnt + 3
          ret_target_list(ret_cnt:) = fld_name
          ret_cnt = ret_cnt + fld_name_len

##      }            /* End of FORMDATA loop */
        Get_Form_Data = ret_stat
        return
##      end
```

Chapter 5: Embedded QUEL for Ada

This chapter describes the use of EQUEL with the Ada programming language.

EQUEL Statement Syntax for Ada

This section describes the language-specific ground rules for embedding QUEL database and forms statements in an Ada program. An EQUEL statement has the following general syntax:

*EQUEL_statement*

For information on QUEL statements, see the *QUEL Reference Guide*. For information on EQUEL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe how to use the various syntactical elements of EQUEL statements as implemented in Ada.

Margin

There are no specified margins for EQUEL statements in Ada. Always place the two number signs (**##)** in the first two positions of the line. The rest of the statement can begin anywhere else on the line.

Terminator

No statement terminator is required for EQUEL/Ada statements. It is conventional not to use a statement terminator in EQUEL statements, although the Ada statement terminator, the semicolon (;), is allowed at the end of EQUEL statements. The preprocessor ignores it.

For example, the following two statements are equivalent:

```
## sleep 1
```

and

```
## sleep 1;
```

The terminating semicolon may be convenient when entering code directly from the terminal using the **-s** flag. For information on using the **-s** flag to test the syntax of a particular EQUEL statement, see Precompiling, Compiling and Linking an EQUEL Program in this chapter.

EQUEL statements that are made up of a few other statements, such as a **display** loop, only allow a semicolon after the last statement. For example:

```
## display empform            --No semicolon here
## initialize                 --No semicolon here
## activate menuitem "Help"   --No semicolon here
## {
##     message "No help yet"; --Semicolon allowed
##     sleep 2;               --Semicolon allowed
## }
## finalize;                  --Semicolon allowed on last statement
```

Variable declarations made visible to EQUEL observe the normal Ada declaration syntax. Thus, variable declarations must be terminated in the normal way for Ada, with a semicolon.

Line Continuation

There are no special line-continuation rules for EQUEL/Ada. EQUEL statements can be broken between words and continued on any number of subsequent lines. An exception to this rule is that you cannot continue a statement between two words that are reserved when they appear together, such as **declare cursor**. See the *QUEL Reference Guide* for a list of double keywords. Each continuation line must be started with ## characters. Blank lines are permitted between continuation lines.

If you want to continue a character-string constant across two lines, end the first line with a backslash character (\), and continue the string at the beginning of the next line. In this case, do not place ## characters at the beginning of the continuation lines.

For examples of string continuation, see String Literals in this chapter.

Comments

Two kinds of comments can appear in an EQUEL program, EQUEL comments and host language comments. EQUEL comments are delimited by two hyphens (- -), and continue till the end of the line, or by /* and */, and can continue over multiple lines.

Both styles of comments appear on lines beginning with the ## sign. Whereas the preprocessor passes Ada comments through as part of its output, it strips EQUEL comments and does not pass them through. Thus, source code comments that you desire in the preprocessor output should be entered as Ada comments, on lines other than EQUEL lines.

The following restrictions apply to any comments in an EQUEL/Ada program, whether intended as EQUEL comments or Ada comments:

- If anything other than ## appears in the first two positions of a line of EQUEL source, the precompiler treats the line as host code and ignores it. The only exception to this is a string-continuation line (see String Literals in this chapter).
- Comments cannot appear in string constants. In this context, the intended comment will be interpreted as part of the string constant.
- In general, EQUEL comments are allowed in EQUEL statements wherever a space may legally occur. However, no comments can appear between two words that are reserved when they appear together, such as **declare cursor**. Please refer to the list of EQUEL reserved words in the *QUEL Reference Guide.*

The following additional restrictions apply only to Ada comments:

- Ada comments cannot appear between component lines of EQUEL block-type statements. These include **retrieve**, **initialize**, **activate**, **unloadtable**, **formdata**, and **tabledata**, all of which have optional accompanying blocks delimited by open and close braces. Ada comment lines must not appear between the statement and its block-opening delimiter.

  For example:

```
## retrieve (ename = employee.name)
   --Illegal to put a host comment here!
## {
    --A host comment is perfectly legal here
    put ("Employee name: "& ename);
## }
```

- Ada comments cannot appear between the components of compound statements, in particular the **display** statement. It is illegal for an Ada comment to appear between any two adjacent components of the **display** statement, including **display** itself and its accompanying **initialize**, **activate**, and **finalize** statements.

  For example:

```
## display empform
--Illegal to put a host comment here!
## initialize (empname = "Fred McMullen")
--Host comment illegal here!
## activate menuitem "Clear"
## {
--Host comment here is fine
##     clear field all
## }
--Host comment illegal here!
## activate menuitem "End"
## {
##     breakdisplay
## }
--Host comment illegal here!
## finalize
```

  The *QUEL Reference Guide* specifies these restrictions on a statement-by-statement basis.

- On the other hand, EQUEL comments are legal in the locations described in the previous paragraph, as well as wherever a host comment is legal. For example:

```
## retrieve (ename = employee.name)
##--This is an EQUEL comment, legal in this
##--location and it can span multiple lines
## {
        put ("Employee name: "& ename);
## }
```

String Literals

You double quotes to delimite string literals in EQUEL/Ada.

To embed the double quote in a string literal, you should use two double quotes, as in:

```
## message "A quote "" in a string"
```

The Ada single quote character delimiter is also accepted by the preprocessor and is converted to a double quote.

To embed the backslash character, precede it with another backslash.

When continuing an EQUEL statement to another line in the middle of a string literal, use a backslash (****) immediately prior to the end of the first line. In this case, the backslash and the following newline character are ignored by the preprocessor, so that the following line can continue both the string and any further components of the EQUEL statement. Any leading spaces on the next line are considered part of the string. For example, the following is a legal EQUEL statement:

```
## message "Please correct errors found in updating \
the database tables."
```

Note that you cannot use the Ada concatenation operator (**&**) to continue string literals on the next line.

Block Delimiters

EQUEL block delimiters mark the beginning and end of the embedded block-structured statements. The **retrieve** loop and the forms statements **display**, **unloadtable**, **submenu**, **formdata**, and **tabledata** are examples of block-structured statements. The block delimiters to such statements can be braces, **{** and **}**, or the keywords **begin** and **end**. For example:

```
## display empform
## activate menuitem "Help"
## {
            Help_File("empform");
## }
## activate menuitem "Quit"
## begin
##     breakdisplay
## end
```

Ada Variables and Data Types

This section describes how to declare and use Ada program variables in EQUEL.

Variable and Type Declarations

The following sections describe Ada variable and type declarations.

EQUEL Variable Declarations Procedures

EQUEL statements use Ada variables to transfer data from a database or a form into the program and conversely. You must declare Ada variables to EQUEL before using them in EQUEL statements. Ada variables are declared to EQUEL by preceding the declaration with the ## mark. The declaration must be in a syntactically correct position for the Ada language.

In general, each declared object can be referred to in the scope of the enclosing compilation unit. An object name cannot be redeclared in the same compilation unit scope. For details on the scope of types and variables, see Compilation Units and the Scope of Variables in this chapter.

The With Equel and With Equel_Forms Statements

Along with your declarations, and prior to any executable EQUEL statements or Ada compilation units in your file, you must issue the following Ada **with** statement:

```
## with equel;
```

If the compilation unit uses EQUEL/FORMS statements, you should instead issue the statement:

```
## with equel_forms;
```

The above statements instruct the preprocessor to generate code to call Ingres runtime libraries. Both statements generate Ada **with** and **use** statements to make all the generated calls acceptable to the Ada compiler by including their package specifications. Note that both statements *must* terminate with a semicolon, as required by Ada.

The EQUEL and EQUEL_FORMS package specifications should already be in your Ada program library. (For the appropriate procedures, see Entering EQUEL Package Specifications in this chapter.) Both packages assume that the types **integer**, **float**, **string**, and **address** have not been redefined by any other packages or type declarations included in your file.

Reserved Words in Declarations and Program Units

All EQUEL keywords are reserved. You cannot declare variables with the same names as EQUEL keywords. You can only use them in quoted string literals. These words are:

access	**declare**	**new**	**record**	**type**
array	**delta**	**others**	**renames**	**when**
body	**digits**	**package**	**return**	**use**
case	**function**	**private**	**separate**	
constant	**limited**	**procedure**	**subtype**	

Data Types and Constants

The EQUEL/Ada preprocessor accepts certain data types and constants from the Ada STANDARD and SYSTEM packages. The table below maps the types to their corresponding Ingres type categories. For information on the exact type mapping, see Data Type Conversion in this chapter.

Ada Data Types and Corresponding Ingres Types

Ada Type	Ingres Type
short_short_integer	**integer**
short_integer	**integer**
integer	**integer**
natural	**integer**
positive	**integer**
boolean	**integer**
float	**float**
long_float	**float**
f_float	**float**
d_float	**float**
character	**character**
string	**character**

None of the types listed above should be redefined by your program. If they are redefined, your program may not compile and will not work correctly at runtime.

The table below maps the Ada constants to their corresponding Ingres type categories.

Ada Constants and Corresponding Ingres Types

Ada Constant	Ingres Type
max_int	**integer**
min_int	**integer**
true	**integer**
false	**integer**

Note that, if the type or constant is derived from the SYSTEM package, the program unit must specify that the SYSTEM package should be included—EQUEL does not do so itself. You cannot refer to a SYSTEM object by using the package name as a prefix, because EQUEL does not allow this type of qualification. The types **f_float** and **d_float** and the constants **max_int** and **min_int** are derived from the SYSTEM package.

The Integer Data Type

All **integer** types and their derivatives are accepted by the preprocessor. Even though some integer types do have Ada constraints, such as the types **natural** and **positive**, EQUEL does not check these constraints, either during preprocessing or at runtime. An **integer** constant is treated as an EQUEL constant value and cannot be the target of a Ingres assignment.

The type **boolean** is handled as a special type of **integer**. In Ada, the **boolean** type is defined as an enumerated type with enumerated literals **false** and **true**. EQUEL treats the **boolean** type as an enumerated type and generates the correct code in order to use this type to interact with a Ingres integer. Enumerated types are described in more detail later.

The Float Data Type

There are four floating-point types that are accepted by the preprocessor. The types **float** and **f_float** are the 4-byte floating-point types. The types **long_float** and **d_float** are the 8-byte floating-point types. **Long_float** requires some extra definitions or default Ada pragmas to be able to interact with Ingres floating-point types. Note that the preprocessor does not accept **long_long_float** and **h_float** data types.

The Long Float Storage Format

Ingres requires that the storage representation for long floating-point variables be **d_float**, because the EQUEL runtime system uses that format for floating-point conversions. If your EQUEL program has **long_float** variables that interact with the EQUEL runtime system, you must make sure they are stored in the **d_float** format. Floating-point values of types **g_float** and **h_float** are stored in different formats and sizes. The default Ada format is **g_float**; consequently, you must convert your long floating-point variables to type **d_float**. You can use three methods to ensure that the Ada compiler always uses the **d_float** format.

The first method is to issue the following Ada pragma before every compilation unit that declares **long_float** variables:

```
pragma long_float( d_float );
...
## dbl: long_float;
```

Note that the **pragma** statement is not an EQUEL statement, but an Ada statement that directs the compiler to use a different storage format for **long_float** variables.

The second method is a more general instance of the first. If you are certain that all **long_float** variables in your Ada program library will use the **d_float** format, including those not interacting with Ingres, then you can install the pragma into the program library by issuing the following ACS command:

```
$ acs set pragma/long_float=d_float
```

This system-level command is equivalent to issuing the Ada **pragma** statement for each file that uses **long_float** variables.

The third method is to use the type **d_float** instead of the type **long_float**. This has the advantage of allowing you to mix both **d_float** and **g_float** storage formats in the same compilation unit. Of course, all EQUEL floating-point variables must be of the **d_float** type and format. For example:

```
## d_dbl: d_float;
   g_dbl: g_float; -- Unknown to EQUEL
```

One side effect of all the above conversions is that some default system package instantiations for the type **long_float** become invalid, because they are set up under the **g_float** format. For example, the package LONG_FLOAT_TEXT_IO, which is used to write long floating-point values to text files, must be reinstantiated. Assuming that you have issued the following ACS command on your program library:

```
$ acs set pragma/long_float=d_float
```

you must reinstantiate the LONG_FLOAT_TEXT_IO package before you can use it. A typical file might contain the following two lines, which serve to enter your own copy of LONG_FLOAT_TEXT_IO into your library:

```
with text_io;
package long_float_text_io is new
    text_io.float_io(long_float);
```

A later statement, such as:

```
with long_float_text_io; use long_float_text_io;
```

will pick up your new copy of the package, which is defined using the **d_float** internal storage format.

The Character and String Data Types

Both the **character** and **string** data types are compatible with Ingres string objects. By default, the **string** data type is an array of characters.

The **character** data type does have some restrictions. Because it must be compatible with Ingres string objects, you can use only a one-dimensional array of characters. Therefore, you cannot use a single character or a multidimensional array of characters. Note that you *can* use a multidimensional array of strings.

For example, the following four declarations are legal:

```
## subtype Alphabet is Character range 'a'..'z';
## type word_5 is array(1..5) of Character;
                                    -- 1-dimensional array
## word_6: String(1..6);            -- Default string type
## word_arr: array(1..5) of String(1..6);
                                    -- Array of strings
```

However, the declarations below are illegal, because they violate the EQUEL restrictions for the **character** type. Although the declarations may not generate EQUEL errors, the references will not be accepted by the Ada compiler when used with EQUEL statements. For example:

```
## letter: Character; -- 1 character
## word_arr: array(1..5) of word_5;
                      -- 2-dimensional array of char
```

Both could be declared instead with the less restrictive **string** type:

```
## letter: array(1..1) of Character;-- or equivalently...
## letter: String(1..1);
## word_arr: array(1..5) of String(1..6);   -- Array of strings
```

Variable and Number Declaration Syntax

The following sections describe the syntax for variable and number declarations.

Simple Variable Declarations

An EQUEL/Ada variable declaration has the following syntax:

identifier {, *identifier*} **:**
[**constant**]
[**array** (*dimensions*) of]
type_name
[*type_constraint*]
[**:=** *initial_value*]**;**

Syntax Notes:

1. The *identifier* must be a legal Ada identifier beginning with an alphabetic character.
2. If the **constant** clause is specified, the declaration must include an explicit initialization.
3. If the **constant** clause is specified, the declared variables cannot be targets of Ingres assignments.

4. The *dimensions* of an **array** specification are not parsed by the EQUEL preprocessor. Consequently, unconstrained array bounds and multidimensional array bounds will be accepted by the preprocessor. However, an illegal *dimension* (such as a non-numeric expression) will also be accepted but will cause Ada compiler errors. For example, both of the following declarations are accepted, even though only the first is legal Ada:

```
## square: array (1..10, 1..10) of Integer;
## bad_array: array ("dimensions") of Float;
```

5. The *type_name* must be either an EQUEL/Ada type (see Data Types and Constants) or a type name already declared to EQUEL.

6. The legal *type_constraints* are described in the next section.

7. The *initial_value* is not parsed by the preprocessor. Consequently, any initial value is accepted, even if it may later cause a Ada compiler error. For example, both of the following initializations are accepted, even though only the first is legal Ada:

```
## rowcount: Integer := 1;
## msgbuf: String(1..100) := 2;
        -- Incompatible value
```

You must not use a single quote in an initial value to specify an Ada attribute. EQUEL will treat it as the beginning of a string literal and will generate an error. For example, the following declaration will generate an error:

```
id: Integer := Integer 'first
## rows, records:Integer range 0..500 := 0;
## was_error:    Boolean;
## min_sal:       constant Float := 15000.00;
## msgbuf:         String(1..100) := (1..100 = ' ');
## operators:  constant array(1..6) of String(1..2)
##         := ("= ", "!=", "< ", "> ", "<=", ">=");
```

Type Constraints

Type constraints can optionally follow the type name in an Ada object declaration. In general, they do not provide EQUEL with runtime type information, so they are not fully processed. The following two constraints describe the syntax and restrictions of EQUEL type constraints.

The Range Constraint

The syntax of the range constraint is:

range *lower_bound .. upper_bound*

In a variable declaration, its syntax is:

*identifier***:** *type_name* **range** *lower_bound .. upper_bound***;**

Syntax Notes:

1. Even if a range constraint is not allowed by Ada, it will be accepted by EQUEL. For example, both of the following range constraints are accepted, although the second is illegal in Ada because the **string** type is not a discrete scalar type:

```
## digit: Integer range 0..9;
## chars: String range 'a'..'z';
```

2. The two bounds, *lower_bound* and *upper_bound*, must be integer literals, floating-point literals, character literals, or identifiers. Other expressions are not accepted.

3. The bounds are not checked for compatibility with the *type_name* or with each other. For example, the following three range constraints are accepted, even though only the first is legal Ada:

```
## byte: Integer range -128..127;
## word: Integer range 1.0..30000.0;
                                   -- Incompatible with type name
## long: Integer range 1..'z';   -- Incompatible  with each other
```

The Discriminant and Index Constraints

The discriminant and index constraints have the following syntax:

(discriminant_or_index_constraint)

In a variable declaration the syntax is:

identifier**:** *type_name (discriminant_or_index_constraint);*

Syntax Notes:

1. Even if a constraint is not allowed by Ada, it will be accepted by EQUEL. For example, both of the following constraints are accepted, even though the second is illegal in Ada because the **integer** type does not have a discriminant:

```
## who: String(1..20);-- Legal index constraint
## nat: Integer(0);   -- Illegal context for discriminant
```

2. The contents of the constraint contained in the parentheses are not processed. Consequently, any constraint will be accepted, even if not allowed by Ada. For example, the following declaration will be accepted by EQUEL but will generate a later Ada compiler error because of the illegal index constraint:

```
## password: String(secret word);
```

Note that the above type constraints are not discussed in detail after this section, and their rules and restrictions are considered part of the EQUEL/Ada declaration syntax.

Formal Parameter Declarations

An EQUEL/Ada formal parameter declaration has the following syntax:

identifier {**,** *identifier*} **:**
[**in** | **out** | **in out**]
type_name
[:= *default_value*]
[;]

In a subprogram declaration, its syntax is:

procedure *name* **(** *parameter_declaration* {**;** *parameter_declaration*} **)**
is
...

or

function *name* **(** *parameter_declaration* {**;** *parameter_declaration*} **)**
return *type_name* **i**s
...

Syntax Notes:

1. If the **in** mode alone is specified, the declared parameters are considered constants and cannot be targets of Ingres assignments.
2. If no mode is specified, the default **in** mode is used and the declared parameters are considered constants.
3. The *type_name* must be either an EQUEL/Ada type or a type name already declared to EQUEL.
4. The *default_value* is not parsed by the preprocessor. Consequently, any default value is accepted, even if it may cause a later Ada compiler error. For example, both of the following parameter defaults are accepted, even though only the first is legal in Ada:

   ```
   ## procedure Load_Table
   ##   (clear_it: in Boolean := TRUE;
   ##   is_error: out Boolean := "FALSE") is
    ...
   ```

   You must not use a single quote in a default value to specify an Ada attribute. EQUEL will treat it as the beginning of a string literal and will generate an error.
5. The semicolon is required with all parameter declarations except the last.
6. The scope of the parameters is the subprogram in which they are declared. For detailed scope information, see Compilation Units and the Scope of Variables in this chapter.

Number Declarations

An EQUEL/Ada number declaration has the following syntax:

identifier {**,** *identifier*} **:**
constant:= *initial_value****;***

Syntax Notes:

1. A number declaration is only allowed for integer numbers. You cannot declare a floating-point number declaration using this format. If you do, EQUEL will treat it as an integer number declaration, causing later compiler errors. For example, the following two number declarations are treated as integer number declarations, even though the second is a float number declaration:

   ```
   ## max_employees: constant := 50000;
   ## min_salary: constant := 13500.0; --Treated as INTEGER
   ```

   To declare a constant float declaration, you must use the **constant** variable syntax. For example, the second declaration above should be declared as:

   ```
   ## min_salary: constant Float := 13500.0;  -- Treated as FLOAT
   ```

2. The declared numbers cannot be the targets of Ingres assignments.
3. The *initial_value* is not parsed by the preprocessor. Consequently, any initial value is accepted, even if it may later cause an Ada compiler error. For example, both of the following initializations are accepted, even though only the first is a legal Ada number declaration:

   ```
   ## no_rows: constant := 0;
   ## bad_num: constant := 123 + "456";
   ```

You must not use a single quote in an initial value to specify an Ada attribute. EQUEL will treat it as the beginning of a string literal and will generate an error.

Renaming Variables

The syntax for renaming variables is:

*identifier***:** *type_name* **renames** *declared_object***;**

Syntax Notes:

1. The *type_name* must be a predeclared type, and the *declared_object* must be a known EQUEL variable or constant.
2. The *declared_object* must be compatible with the *type_name* in base type, array dimensions and size.

3. If the declared object is a record component, any subscripts used to qualify the component are ignored. For example, both of the following **rename** statements will be accepted even though one of them must be wrong, depending on whether "emprec" is an array:

```
## eage1: Integer renames emprec(2).age;
## eage2: Integer renames emprec.age;
```

Type Declaration Syntax

EQUEL/Ada supports a subset of Ada type declarations. In a declaration, the EQUEL preprocessor only notes semantic information relevant to the use of the variable in EQUEL statements at runtime. Other semantic information is ignored by the preprocessor. Refer to the syntax notes in this section and its subsections for details.

Type Definition

An EQUEL/Ada full type declaration has the following syntax:

type *identifier* [*discriminant_part*] **is** *type_definition****;***

Syntax Notes:

1. The *discriminant_part* has the syntax:

   (*discriminant_specifications***)**

   and is not processed by EQUEL. As with variable declarations, a discriminant specification will always be accepted by EQUEL, even if not allowed by Ada. For example, the following declaration will be accepted by EQUEL but will later generate an Ada compiler error, because the discriminant type is not a discrete type and the discriminant part is not allowed in a non-record declaration:

   ```
   ## type shapes(name: String := "BOX")
   ##      is array(1..10) of String(1..3);
   ```

   From this point on, discriminant parts are not included in the syntax descriptions or notes.

2. The legal *type_definitions* allowed in type declarations are described below.

Subtype Definition

An EQUEL/Ada **subtype** declaration has the following syntax:

subtype *identifier* **is** *type_name* [*type_constraint*]**;**

Syntax Note:

The *type_constraint* has the same rules as the type constraint of a variable declaration. The range, discriminant, and index constraints are all allowed and are not processed against the *type_name* being used. For more details about these constraints, refer to the section above on variable type constraints. The floating-point constraint and the **digits** clause, which are allowed in subtype declarations, are discussed later.

Integer Type Definitions

The syntax of an EQUEL/Ada integer type definition is:

range *lower_bound .. upper_bound*

In the context of a type declaration, the syntax is:

type *identifier* **is range** *lower_bound .. upper_bound;*

In the context of a subtype declaration, the syntax is:

subtype *identifier* **is** *integer_type_name*
range *lower_bound .. upper_bound;*

Syntax Notes:

1. In an integer type declaration (not a subtype declaration), the range constraint of an integer type definition is processed by EQUEL to evaluate storage size information. Both *lower_bound* and *upper_bound* must be integer literals. Based on the specified range and the actual values of the bounds, EQUEL treats the type as a byte-size, a word-size, or a longword-size integer. For example:

```
## type Table_Num is range 1..200;
```

2. In an integer subtype declaration, the range constraint is treated as a variable range constraint and is not processed. Consequently, the same rules that apply to range constraints for variable declarations apply to integer range constraints for integer subtype declarations. The base type and storage size information is determined from the *integer_type_name* used. For example:

```
## subtype Ingres_I1 is Integer range -128..127;
## subtype Ingres_I2 is Integer range -32768..32767;
## subtype Table_Low is Table_Num range 1..10;
## subtype Null_Ind  is Short_Integer range -1..0;   -- Null Indicator
```

Floating-point Type Definitions

The syntax of an EQUEL/Ada floating-point type definition is:

digits *digit_specification* [*range_constraint*]

In the context of a type declaration the syntax is:

type *identifier* **is digits** *digit_specification* [*range_constraint*]**;**

The syntax of a floating-point subtype declaration is:

subtype *identifier* **is** *floating_type_name*
[**digits** *digit_specification*]
[*range_constraint*]**;**

Syntax Notes:

1. The value of *digit_specification* must be an integer literal. Based on the value of the specification, EQUEL will determine whether to treat a variable of that type as a 4-byte float or an 8-byte float. The rules in the following table are applicable.

Digit Range	Type
1 < = *d* < = 6	4-byte floating-point type
7 < = *d* < = 16	8-byte floating-point type

   Note that if the digits specified are out of range, the type is unusable. Recall that EQUEL does not accept either the **long_long_float** or the **h_float** type. For detailed information on the internal storage format for 8-byte floating-point variables, see The Long Float Storage Format in this chapter.

2. The *range_constraint* for floating-point types and subtypes is treated as a variable range constraint and is not processed. Although EQUEL will allow any range constraint, you should not specify a range constraint that will alter the size needed to store the declared type. EQUEL obtains its type information from the **digits** clause, and altering this type information by a range clause, which may require more precision, will result in runtime errors.

3. The **digits** clause in a subtype declaration does not have any effect on the EQUEL type information. This information is obtained from *floating_type_name*. For example:

```
## type Emp_Salary   is digits 8 range
                    0.00..500000.00;

## subtype Directors_Sal is Emp_Salary
                    100500.00..500000.00;
## subtype Raise_Percent is Float range 1.05..1.20;
```

Enumerated Type Definitions

The syntax of an EQUEL/Ada enumerated type definition is:

(*enumerated_literal* {**,** *enumerated_literal*}**)**

In the context of a type declaration, the syntax is:

type *identifier* **is** (*enumerated_literal* {**,** *enumerated_literal*}**);**

In the context of a subtype declaration, the syntax is:

subtype *identifier* **is** *enumerated_type_name* [*range_constraint*]**;**

Syntax Notes:

1. There can be at most 1,000 enumerated literals in an enumerated type declaration. The preprocessor treats all literals and variables declared with this type as integers. Enumerated literals are treated as though they were declared with the **constant** clause, and therefore they cannot be the targets of Ingres assignments. When an enumerated literal is used with embedded statements, only the ordinal position of the value in relation to the original enumerated list is relevant. When assigning from an enumerated variable or literal, the preprocessor generates:

   *enumerated_type_name***'pos(***enumerated_literal***)**

   When assigning from or into an enumerated variable, the preprocessor passes the object by address and assumes that the value being assigned from or into the variable will not raise a runtime constraint error.

2. An enumerated literal can be an identifier or a character literal. EQUEL does not store or process enumerated literals that are character literals.

3. Enumerated literal identifiers must be unique in their scope. EQUEL does not allow the overloading of variables or constants.

4. The *range_constraint* for enumerated subtypes is treated as a variable range constraint and is not processed. The type information is determined from *enumerated_type_name*. For example:

```
## type Table_Field_States is
## (UNDEFINED, NEWROW, UNCHANGED, CHANGED, DELETED);
## subtype Updated_States is Table_Field_States
## range CHANGED..DELETED;
## tbstate: Table_Field_States := UNDEFINED;
```

5. EQUEL accepts the predefined enumeration type name **boolean** that contains the two literals FALSE and TRUE.

6. You can use a representation clause for enumerated types. When you do so, however, you should not reference any enumerated literals of that type in the embedded statements. Enumerated literals are interpreted into their integer relative position (**pos**) and representation clauses invalidate the effect of the **pos** attribute that the preprocessor generates. The representation clauses should not be preceded by the ## mark.
7. Enumerated variables and literals can only be used to assign to or from Ingres. These objects cannot be used to specify simple numeric objects, such as table field row numbers or **sleep** statement seconds.

Array Type Definitions

The syntax of an EQUEL/Ada array type definition is:

array (*dimensions***) of** *type_name;*

In the context of a type declaration, the syntax is:

type *identifier* **is array (***dimensions***) of**
type_name [*type_constraint*]**;**

Syntax Notes:

1. The *dimensions* of an **array** specification are not parsed by the EQUEL preprocessor. Consequently, unconstrained array bounds and multidimensional array bounds will be accepted by the preprocessor. However, an illegal dimension (such as a non-numeric expression) will also be accepted but will later cause Ada compiler errors. For example, both of the following type declarations are accepted, even though only the first is legal in Ada:

   ```
   ## type Square is array(1..10, 1..10) of Integer;
   ## type What is array("dimensions") of Float;
   ```

   Because the preprocessor does not store the array dimensions, it only checks to determine that when the array variable is used, it is followed by a subscript in parentheses.
2. The *type_constraint* for **array** types is treated as a variable type constraint and is not processed. The type information is determined from *type_name*.
3. Any array built from the base type **character** (*not* **string**) must be exactly one-dimensional. EQUEL will treat the whole array as though it were declared as type **string**. If more dimensions are declared for a variable of type **character**, EQUEL will still treat it as a one-dimensional array.
4. The type **string** is the only array type.

Record Type Definitions

The syntax of an EQUEL/Ada record type definition is:

record
record_component {*record_component*}
end record;

where *record_component* is:

*component_declaration***;** | *variant_part***;** | **null;**

where *component_declaration* is:

identifier {**,** *identifier*} **:**
type_name [*type_constraint*] [:= *initial_value*]

In the context of a type declaration, the syntax of a record type definition is:

type *identifier is*
record
record_component { *record_component*}
end record;

Syntax Notes:

1. In a *component_declaration*, all clauses have the same rules and restrictions as they do in a regular type declaration. For example, as in regular declarations, the preprocessor does not check initial values for correctness.
2. The *variant_part* accepts the Ada syntax for variant records: if specified, it must be the last component of the record. The variant discriminant name, choice names, and choice ranges are all accepted. There is no syntactic or semantic checking on those variant objects. EQUEL uses only the final component names of the variant part and not any of the variant object names.
3. You can specify the **null** record.
4. A *record_component* can also be Ada host code. Consequently, you can include components that will not be used by EQUEL (and with types unknown to EQUEL), by not marking the line with a ## mark. Also, if some *variant_part* syntaxes are not accepted by EQUEL, you do not have to mark those lines, as EQUEL does not store the variant information. For example:

```
## type Address_Rec is
## record
##    street: String(1..30);
##    town: String(1..10);
##    zip: Positive;
## end record;
```

```
## type Employee_Rec is
## record
##    name:   String(1..20);
##    age:    Short_Short_Integer;
##    salary: Float := 0.0;
##    address: Address_Rec;
-- The following two components are unknown to EQUEL
        scale:   Long_Long_Float;
        checked: Boolean := FALSE;
## end record;
```

Incomplete Type Declarations and Access Types

The incomplete type declaration should be used with an access type. The syntax for an incomplete type declaration is:

type *identifier* [*discriminant_part*];

Syntax Notes:

1. As with other type declarations, the *discriminant_part* is ignored.
2. You must fully define an incomplete type before using any object declared with it.

The syntax for an access type declaration is:

type *identifier* **is access** *type_name* [*type_constraint*]**;**

Syntax Notes:

1. The *type_name* must be a predeclared type, whether it is a full type declaration or an incomplete type declaration.
2. The *type_constraint* has the same rules as other type declarations.

The following is an example of the incomplete type declaration:

```
## type Employee_Rec; -- Incomplete declaration
## type Employee is access
##      Employee_Rec;-- Access to above

## type Employee_Rec is -- Real definition
## record
##    name: String(1..20);
##    age: Short_Short_Integer;
##    salary: Float := 0.0;
##    link: Employee;
## end record;
```

Derived Types

The syntax for a derived type is:

type *identifier* **is new** *type_name* [*type_constraint*]**;**

Syntax Notes:

1. The *type_name* must be a predeclared type, whether it is a full type declaration or an incomplete type declaration.
2. EQUEL assigns the type being declared the same properties as the *type_name* specified. The preprocessor will make sure that any variables of a derived type are cast into the original base type when used with the runtime routines.
3. The *type_constraint* has the same rules as other type declarations.

The following example illustrates the use of the derived type:

```
##  type Dbase_Integer is new Integer;
```

Private Types

The syntax for a private type is:

type *identifier* **is** [**limited**] **private;**

Syntax Note:

This type declaration is treated as an incomplete type declaration. You must fully define a private type before using any object declared with it.

Representation Clauses

With one exception, you must not use the representation clause for any types or objects you have declared to EQUEL and intend to use with the EQUEL runtime system. Any such clause will cause runtime errors. These clauses include the Ada statement:

for *type_or_attribute* **use** *expression**;***

and the Ada pragma:

pragma pack(*type_name***);**

The exception is that you can use a representation clause to specify internal values for enumerated literals. When you do so, however, you should not reference any enumerated literals of the modified enumerated type in embedded statements. The representation clause invalidates the effect of the **pos** attributes that the preprocessor generates. If the application context is one that requires the assignment from the enumerated type, then you should deposit the literal into a variable of the same enumerated type and assign that variable to Ingres. In all cases, do not precede the representation clause with the ## mark.

For example:

```
    ## type Opcode is (OPADD, OPSUB, OPMUL);
    for Opcode use (OPADD => 1, OPSUB => 2,
                                OPMU, =>4);
    ...
        opcode_var ;= OPSUB;
## append to codes (opcode = opcode_var);
```

Indicator Variables

An *indicator variable* is a 2-byte integer variable. There are three possible ways to use these in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a null.
- In a statement that sets data to Ingres, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable is large enough to hold the full length of the returned character string.

To declare an indicator variable you should use the **short_integer** data type. The following example declares two indicator variables:

```
## ind: Short_Integer; -- Indicator variable
## ind_arr: array(1..10) of Short_Integer; -- Indicator array
```

Note that a variable declared with any derivative of the **short_integer** data type will be accepted as an indicator variable.

Assembling and Declaring External Compiled Forms

You can pre-compile your forms in the Visual Forms Editor (VIFRED). This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. After the file is created the file, use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the EQUEL/FORMS statement **addform** can refer to this global object, it must be declared in an EQUEL declaration section. The Ada compiler requires that the declaration be in a package and that the objects be imported with the **import_object** pragma.

The syntax for a compiled form package is:

```
##    package compiled_forms_package is
##       formname: Integer;
                    pragma import_object( formname );
##    end compiled_forms_package;
```

You must then issue the Ada **with** and **use** statements on the compiled form package before every compilation unit that refers to the form:

```
with compiled_forms_package; use compiled_forms_package;
```

Syntax Notes:

1. The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other EQUEL/FORMS statements. In all statements that use *formname* as the form title you must dereference the name with a # sign.
2. The **import_object** pragma associates the object with the external form definition. In order to use this pragma, the package must be issued in the outermost scope of the file.

The next example shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
## package Compiled_Forms is
## empform: Integer;
        pragma import_object( empform );
## end Compiled_Forms;
        ...

with Compiled_Forms; use Compiled_Forms;
        ...

## addform empform; -- The imported object
## display #empform; -- The name of the form
        ...
```

Concluding Example

The following example demonstrates some simple EQUEL/Ada declarations:

```
## package Compiled_Forms is
## empform, deptform: Integer; -- Compiled forms

pragma import_object( empform );
pragma import_object( deptform );
## end Compiled_Forms;

with Compiled_Forms; use Compiled_Forms;

## package Concluding_Example is

## MAX_PERSONS: constant := 1000;
## dbname: String(1..9) := "personnel";
## formname, tablename, columnname: String(1..12);
```

```
## salary: Float;

## type DATATYPES_REC is -- Structure of all types
##   d_byte:   Short_Short_Integer;
##   d_word:   Short_Integer;
##   d_long:   Integer;
##   d_single: Float;
##   d_double: Long_Float;
##   d_string: String(1..20);
## end record;
## d_rec: DATATYPES_REC;

## -- Record with a discriminant
## record PERSONTYPE_REC (married: in Boolean) is
##   age: Short_Short_Integer;
##   flags: Integer;
##   case married:
##     when TRUE =>
##        spouse_name: String(1..30);
##     when FALSE =>
##        dog_name: String(1..12);
##   end case;
## end record;
## person: PERSONTYPE_REC(TRUE);
## person_store: array(1..MAX_PERSONS) of PERSONTYPE_REC(FALSE);

## ind_var: Short_Integer := -1; -- Indicator Variable

## end Concluding_Examples;
```

Compilation Units and the Scope of Variables

Type names and variable names are local to the closest enclosing Ada compilation unit. EQUEL/Ada compilation units include procedures, functions, package bodies and declaration blocks, all of which can be declared to EQUEL. The objects visible in the scopes include objects that are visible in the parent scope, formal parameters (if applicable) and local declarations. You cannot use the dotted notation to refer to hidden or ambiguous objects by prefixing the object with a subprogram or package name.

As in Ada, once the preprocessor has exited the scope, the variables are no longer visible and cannot be referenced. The Ada package specification is an exception to this visibility rule, because all the package specification contents are visible outside of the package.

The Package Specification

The syntax for an EQUEL/Ada package specification is:

package *package_name* **is**
[*declarations*]
end [*package_name*]**;**

Syntax Notes:

1. *Package_names* on the **package** and **end** statements are not processed and are not compared for equivalence, as required by Ada.
2. You cannot qualify objects in the package specification with any package names.
3. Variables declared in package specifications are global to the parent scope of the specification. This is true even for objects declared in the **private** section.

When EQUEL reads a package specification, no matter whether it is declared in the same file or included by means of the EQUEL **include** statement, the contents of the package become visible immediately afterwards. EQUEL behaves as though there were the implicit Ada statements:

```
with package_name; use package_name;
```

The use of the EQUEL **include** statement actually generates the Ada **with** and **use** clauses, using the file name as the package name. The preprocessor generates these statements and assumes global visibility of package specification contents, because it does not read Ada library units. This restriction indicates that two package specifications declared at the same scope level cannot declare two objects with the same name. Note that when a package specification is nested in another compilation unit or package specification, it does not create a new scope level. The following example will generate an error because of the redeclaration of the object "ptr":

```
## package Stack is
## stack_max: constant := 50;
## ptr: Integer range 1..stack_max;
## stack_arr: array(1..stack_max) of Integer;
## end Stack;

## package Employees is
## ename_arr: array(1..1000) of String(1..20);
## ptr: Integer range 1..1000;
## end Employees;
```

If a package specification declares several types and variables that will be used with various subprograms and package bodies, you should put the specification in a file by itself and use the EQUEL **include** statement. The **include** statement will re-read the original text file and behave as though you had issued the appropriate Ada **with** and **use** clauses. For more information on the EQUEL **include** statement, see Include File Processing in this chapter.

If you do *not* use the EQUEL **include** statement, *you* must explicitly issue the Ada **with** and **use** clauses. The following example declares two variables inside a package specification. In a single file are two procedures, which must both be preceded by the **with** and **use** clauses:

```
## package Vars is
## var1: Integer;
## var2: String(1..3);
## end Vars37
## with Vars; use Vars; -- Explicit Ada visibility clauses
```

```
## procedure Read_Vars is
## begin                -- EQUEL Statements that retrieve var1 and var2
## end Read_Vars;

with Vars; use Vars;    -- Explicit Ada visibility clauses

## procedure Write_Vars is
## begin                -- EQUEL Statements that append var1 and var2
## end Write_Vars;
```

The Package Body

The syntax for an EQUEL/Ada package body is:

package body *package_name* **is**
[*declarations*]
[**begin**
statements]
end [*package_name*]**;**

Syntax Notes:

1. *Package_names* on the **package body** and ***end*** statements are not processed and are not compared for equivalence, as required by Ada.
2. You cannot qualify objects in the package specification with any package names.
3. Variables declared in a package body are visible to the package body and to any nested blocks.
4. If the package body requires knowledge of the package specification, you *must* make the specification known to EQUEL. This can be done either by including the specification's file by means of the EQUEL **include** statement, or by including the text of the specification in the EQUEL source file. EQUEL does not assume knowledge of the package specification with the same name as the body.
5. EQUEL does not process *separate* compilation units and, consequently, does not allow the Ada **separate** clause.

The Procedure

The syntax for an EQUEL/Ada procedure is:

procedure *proc_name* [**(***formal_parameters***)**] **is**
[*declarations*]
begin
statements
end [*proc_name*]**;**

Syntax Notes:

1. *Proc_names* on the **procedure** and **end** statements are not processed and are not compared for equivalence, as required by Ada.
2. Formal parameters and variables declared in a procedure are visible to the procedure and to any nested blocks.
3. Formal parameters and their syntax are described in the section on variable declarations.

The Function

The syntax for an EQUEL/Ada function is:

function *func_name* [**(***formal_parameters***)**] **return** *result_type* **is**
    [*declarations*]
begin
    statements
end [*func_name*]**;**

Syntax Notes:

1. *Func_names* on the **function** and **end** statements are not processed and are not compared for equivalence, as required by Ada.
2. EQUEL need not know the *result_type*, because EQUEL does not allow the use of functions in place of variables in executable statements.
3. Formal parameters and variables declared in a function are visible to the function and to any nested blocks.
4. Formal parameters and their syntax are described in the section on variable declarations.

The Declaration Block

The syntax for an EQUEL/Ada declaration block is:

declare
    declarations
begin
    statements
end [*block_name*]**;**

Syntax Notes:

1. *Block_name* is not processed and is not compared for equivalence against any block labels (if used).
2. Variables declared in a declaration block are visible to the declaration block and to any nested blocks.

Variable and Type Scope

As mentioned above, variables and types are visible in the block in which they are declared, unless they are declared in a package specification, in which case they are globally visible. Variables can be redeclared only in a nested scope, such as in a declaration block or a nested procedure. Variables cannot be redeclared in the same scope. For example, the following two enumerated type declarations in the same scope will cause a redeclaration of the overloaded literal "UNDEFINED":

```
## type Question is (SIMPLE, DIFFICULT, UNDEFINED);
## type Answer is (WRONG, RIGHT, SORT_OF, UNDEFINED);
```

Note that you can declare record components with the same name but different record types. The following example declares two records, each of which has the components "firstname" and "lastname":

```
## type Child is
## record
##     firstname: String(1..15);
##     lastname: String(1..20);
##     age: Integer;
## end record;

## type Some_Childs is array(1..10) of Child;

## type Mother is
## record
##     firstname: String(1..15);
##     lastname: String(1..20);
##     num_child: Integer range 1..10;
##     children: Some_Childs;
## end record;
```

The following example shows several different declarations of the variable "var," illustrating how the same object can be redeclared in nested and parallel scopes, each time referring to a different type:

```
## with equel;

## procedure Proc_A(var: type_1) is

-- Will be used even when this particular "var" is hidden
## proc_a_var: type_1 renames var;

## procedure Proc_B is
##   var: type_2;
## begin
   -- Var is of type_2
## end Proc_B;

## function Func_C(var: type_3) return Integer is
## begin

  -- Var is of type_3
  -- Note that you cannot refer to Proc_A.var
  -- but you can refer to proc_a_var of type_1.
## end Func_C;

## begin
     -- Var is of type_1
```

```
## declare
## var: type_4;
## begin
   -- Var is of type_4;
## end;

-- Var is of type_1

## end Proc_A;
```

Special care should be taken when using variables with a **declare cursor** statement. The variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued, and at that time, evaluates any associated variables. For example, in the following program fragment, even though the variable "number" is valid to the preprocessor at the point of the **declare cursor** statement, it is not a valid variable name for the Ada compiler at the point that the **open** is issued.

```
## package Bad_Cursor is
        --This example contains an error

## procedure Init_Csr is
## number: Integer;
## begin

-- Cursor declaration includes reference to "number"
## declare cursor c1 for
##    retrieve (employee.name, employee.age)
##      where employee.num = number

...

## end Init_Csr;

## procedure Process_Csr is
## ename: String(1..15);
## eage: Integer;
## begin
   -- Opening the cursor evaluates invalid "number"
## open cursor c1

## retrieve cursor c1 (ename, eage)

...

## end Process_Csr;

## end Bad_Cursor;
```

Variable Usage

Ada variables declared to EQUEL can substitute for most elements of EQUEL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. To use an Ada variable in an EQUEL statement, just use its name. To refer to an element, such as a database column, with the same name as a variable, dereference the element by using the EQUEL dereferencing indicator (**#**). As an example of variable usage, the following **retrieve** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idnovar" as an expression in the where clause:

```
## retrieve (namevar = e.name, numvar = e.num)
## where e.idno = idnovar;
```

When referencing a variable, you cannot use an Ada attribute, because the attribute is introduced by a single quote. EQUEL will treat this single quote as the beginning of a string literal and will generate a syntax error.

When referencing a variable, you also cannot use the dotted notation to refer to hidden or ambiguous objects by prefixing the object with a subprogram or package name, even if the package is explicitly declared. EQUEL will generate a syntax error on the qualifying dot.

If, in retrieving from Ingres into a program variable, no value is returned for some reason (for example, no rows qualified in a query), the variable will contain an undefined value.

Various rules and restrictions apply to the use of Ada variables in EQUEL statements. The sections below describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

A simple scalar-valued variable (integer, floating-point or character string) is referred to by the syntax:

simplename

Syntax Notes:

1. If the variable is used to send data to Ingres, it can be any scalar-valued variable, constant or enumerated literal.
2. If the variable is used to receive data from Ingres, it cannot be a variable declared with the **constant** clause, a formal parameter that does not specify the **out**mode, a number declaration, or an enumerated literal.
3. A string variable (a 1-dimensional array of characters) is referenced as a simple variable.

The following program fragment demonstrates a typical message-handling routine that uses two scalar-valued variables, "buffer" and "seconds":

```
## procedure Msg (buffer: String; seconds: Integer) is
## begin
## message buffer
## sleep seconds
## end Msg;
```

A special case of a scalar type is the enumerated type. The preprocessor treats all enumerated literals and any variables declared with an enumerated type as integers. When an enumerated literal is used in an EQUEL statement, only the ordinal position of the value in relation to the original enumerated list is relevant. When assigning from an enumerated literal, the preprocessor generates the following:

enumerated_type_name'**pos(enumerated_literal)**

When assigning from or into an enumerated variable, the preprocessor passes the object by address and assumes that the value being assigned from or into the variable will not raise a runtime constraint error. In order to relax the restriction imposed by the preprocessor on enumerated literal assignments (of enumerated types that have included representation clauses to modify their values), you should assign the literal to a variable of the same enumerated type before using it in an embedded statement. For example, the following enumerated type declares the states of a table field row, and the variable of that type will always receive one of those values:

```
## type Table_Field_States is
## (UNDEFINED, NEWROW, UNCHANGED, CHANGED, DELETED);
## tbstate: Table_Field_States := UNDEFINED;
## ename: String(1..20);
...

## getrow empform employee
##  (ename = name, tbstate = _state);

case tbstate is
    when UNDEFINED =>
    ...

end case;
```

Another example retrieves the value TRUE (an enumerated literal of type **boolean**) into a variable when a database qualification is successful:

```
## found: Boolean;
## qual: String(1..100);
..

## retrieve (found = TRUE) where qual;

if (not found) then
...

end if;
```

Array Variables

An array variable is referred to by the syntax:

*arrayname**(subscript**{**,subscript**}**)*

Syntax Notes:

1. The variable must be subscripted, because only scalar-valued elements (integers, floating-point, and character strings) are legal EQUEL values.
2. When the array is declared, the array bounds specification is not parsed by the EQUEL preprocessor. Consequently, illegal bounds values will be accepted. Also, when an array is referenced, the subscript is not parsed, allowing illegal subscripts to be used. The preprocessor only confirms that an array subscript is used for an array variable. You must make sure that the subscript is legal and that the correct number of indices is used.
3. A character string variable is *not* an array and cannot be subscripted in order to reference a single character or a *slice* of the string. For example, if the following variable were declared:

   ```
   ## abc: String(1..3) := "abc";
   ```

   you could not reference

   ```
   abc(1)
   ```

   to access the character "a." To perform such a task, you should declare the variable as an array of three one-character long strings. For example:

   ```
   ## abc: array(1..3) of String(1..1) := ("a","b","c");
   ```

   Note that variables of the Ada **character** type can only be declared as a one-dimensional array. When a variable of that type is used, it must not be subscripted. In the following example, the loop variable "i" is used as a subscript and need not be declared to EQUEL, as it is not parsed.

   ```
   ## formnames: array(1..3) of String(1..8);
      ...

   for i in 1..3 loop
   ## forminit formnames(i)
      end loop;
   ```

Record Components

The syntax EQUEL uses to refer to a record component is the same as in Ada:

record.component{.component}

Syntax Notes:

1. The last record *component* denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and records, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
-- Assume correct declarations for "employee",
--  "person" and other records.
employee.sal   -- Component of a record
person(3).name -- Component of an element of an
                           array
rec1.mem1.mem2.age -- Deeply nested component
```

2. All record components must be fully qualified when referenced. You can shorten the qualification by using the Ada **renames** clause in another declaration to rename some components or nested records.

The following example uses the array of records "emprec" to load values into the tablefield "emptable" in form "empform."

```
##  type Employee_Rec is
##   record
##    ename: String(1..20);
##    eage: Short_Integer;
##    eidno: Integer;
##    ehired: String(1..25);
##    edept: String(1..10);
##    esalary: Float;
##  end record;
## emprec: array(1..100) of Employee_Rec;
...
for i in 1..100 loop
## loadtable empform emptable
##    (name = emprec(i).ename, age = emprec(i).eage,
##     idno = emprec(i).eidno,
##     hired = emprec(i).ehired,
##     dept = emprec(i).edept,
##     salary = emprec(i).esalary)
end loop;
```

If you want to shorten the reference to the record, you can use the **renames** clause to rename a particular member of the "emprec" array, as in the following example:

```
for i in 1..100 loop
##  declare
##    er: Employee_Rec renames emprec(i);
##  begin
##    loadtable empform emptable
##     (name = er.ename, age = er.eage,
##      idno = er.eidno, hired = er.ehired,
##      dept = er.edept, salary = er.esalary)
##  end;
end loop;
```

Access Variables

An access variable must qualify another object by means of the dot operator, using the same syntax as a record component:

access.reference

Syntax Notes:

1. By the time an access variable is referenced, the type to which it is *pointing* must be fully defined. This is true even for access types that were declared to point at incomplete types.
2. The final object denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays, records or access variables, but the last object referenced must be a scalar value.
3. If an access variable is pointing at a scalar-valued type, then the qualification must include the Ada **.all** clause to refer to the scalar value. If used, the **.all** clause must be the last component in the qualification. For example:

```
## type Access_Integer is access Integer;
## ai: Access_Integer;
  ...
ai := new Integer'(2);
## sleep ai.all
```

In the following example, an access type to an employee record is used to load a linked list of values into the Employee database table:

```
## type Employee_Rec;
## type Emp_Link is access Employee_Rec;
## type Employee_Rec is
##   record
##     ename: String(1..20);
##     eage: Short_Integer;
##     eidno: Integer;
##     enext: Emp_Link;
##   end record;
## elist: Emp_Link;
...
while (elist /= null) loop
##  repeat append to employee
##    (name = @elist.ename, age = @elist.eage,
##     idno = @elist.eidno)
elist := elist.enext;
end loop;
```

Using Indicator Variables

The syntax for referring to an *indicator* variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

host_variable:*indicator_variable*

Syntax Note:

The indicator variable can be a simple variable, an array element or a record component that yields a 2-byte integer (**short_integer**). For example:

```
## ind: Short_Integer; -- Indicator variable
## ind_arr: array(1..10) of Short_Integer; -- Indicator
                                           -- array
var_1:ind_var
var_2:ind_arr(2)
```

Data Type Conversion

An Ada variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character string variables.

Data type conversion occurs automatically for different numeric types, such as from floating-point Ingres database column values into integer Ada variables, and for character strings, such as from varying-length Ingres character fields into fixed-length Ada character string buffers.

Ingres does not automatically convert between numeric and character types. You must use the Ingres type conversion operators, the Ingres **ascii** function, or an Ada conversion procedure for this purpose.

The following table shows the default type compatibility for each Ingres data type. Note that some Ada types do not match exactly and, consequently, may go through some runtime conversion.

Ingres TYPES and Corresponding Ada Data Types

Ingres Type	Ada Type
c(*N*), **char**(*N*)	**string**(1..*N*)
c(*N*), **char**(*N*)	**array**(1..*N*) **of character**
text(*N*), **varchar**(*N*)	**string**(1..*N*)
text(*N*), **varchar**(*N*)	**array**(1..*N*) **of character**
i1, integer1	**short_short_integer**
i2, integer2	**short_integer**
i4, integer4	**integer**
f4, float4	**float**
f4 , float4	**f_float**

Ingres Type	Ada Type
f8, float8	**long_float**
f8, float8	**d_float**
date	**string(1..25)**
money	**long_float**

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and forms system and numeric Ada variables. The standard type conversion rules (according to standard VAX rules) are followed. For example, if you assign a **float** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion.

The Ingres **money** type is represented as **long_float**, an 8-byte floating-point value.

Runtime Character Type Conversion

Automatic conversion occurs between Ingres character string values and Ada character string variables. There are four string-valued Ingres objects that can interact with character string variables. They are Ingres names, such as form and column names, database columns of type **c**, **char**, **text** or **varchar**, and form fields of type **character**. Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of Ada character string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string literals "empform" and "empform" refer to the same form.

The conversion of other Ingres objects is a little more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **c** or **character**, a database column of type **text** or **varchar**, or a **character** form field. Ingres pads columns of type **c** or **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **text** or **varchar** or in form fields.

Second, EQUEL assumes that the convention is to blank-pad fixed-length character strings. Character string variables not blank-padded may be storing ASCII nulls or data left over from a previous assignment. For example, the character string "abc" may be stored in an Ada **string(1..5)** variable as the string "abc " followed by two blanks.

When character data is retrieved from a Ingres database column or form field into an Ada character string variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data.

When inserting character data into an Ada Ingres database column or form field from an Ada variable, note the following conventions:

- When data is inserted from an Ada variable into a database column of type **c** or **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from an Ada variable into a database column of type **text** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **text** or **varchar** column. For example, when a string "abc" stored in an Ada **string(1..5)** variable as "abc " (refer to above) is inserted into the **text** or **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the EQUEL **notrim** function. It has the following syntax:

  notrim(*stringvar***)**

  where *stringvar* is a character string variable. An example demonstrating this feature follows later. When used with **repeat** queries, the **notrim** syntax is:

  @notrim(*stringvar***)**

  If the text or varchar column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from an ADA variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

  When comparing character data in an Ingres database column with character data in an ADA variable, note the following convention:
- When comparing data in **c**, **character**, or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Trailing blanks are significant in **text**. Initial and embedded blanks are significant in **character**, **text**, and **varchar**; they are ignored in **c**.

As described above, the conversion of character string data between Ingres objects and ADA variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. For information on the significance of blanks when comparing with various Ingres character types, see the *QUEL Reference Guide*.

The Ingres **date** data type is represented as a 25-byte character string.

The program fragment in the following example demonstrates the **notrim** function and the truncation rules explained above.

```
-- Assume that a table called "textchar" has been created with
-- the following CREATE statement:

--        CREATE textchar
--               (row  = integer4,
--                data = text(10)) -- Note the text data type

##     with EQUEL;
       ...

##     row:  Integer;
##     data: String(1..7) := (1..7 => ' ');
       ...

       data(1..3) := "abc    ";  -- Holds "abc" followed by 4 blanks

       -- The following APPEND adds the string "abc" (blanks truncated)

##     APPEND TO textchar (#row = 1, #data = data)

       -- This statement adds the string "abc    ", with 4 trailing
       -- blanks left intact by using the NOTRIM function.

##     APPEND TO textchar (#row = 2, #data = notrim(data))

       -- This RETRIEVE will retrieve row #2, because the NOTRIM
       -- function left trailing blanks in the "data" variable
       -- in the last APPEND statement.

##     RETRIEVE (row = textchar.#row)
##              WHERE length (textchar.#data) = 7

       put("Row found = ");
       put(row);
```

Dynamically Built Param Statements

The **param** feature dynamically builds EQUEL statements. EQUEL/Ada does not currently support **param** versions of statements. **Param** statements are supported in EQUEL/C and EQUEL/Fortran.

Runtime Error Processing

This section describes a user-defined EQUEL error handler.

Programming for Error Message Output

By default, all Ingres and forms system errors are returned to the EQUEL program, and default error messages are printed on the standard output device. As discussed in the *QUEL Reference Guide*, you can also detect the occurrences of errors by means of the program using the **inquire_ingres** and **inquire_frs** statements. (Use the latter for checking errors after forms statements. Use **inquire_ingres** for all other EQUEL statements.)

This chapter discusses an additional technique that enables your program not only to detect the occurrences of errors, but also to suppress the printing of default Ingres error messages if you choose. The **inquire** statements detect errors but do not suppress the default messages.

This alternate technique entails creating an error-handling function in your program and passing its address to the Ingres runtime routines. Then Ingres will automatically invoke your error handler whenever a Ingres or a forms-system error occurs.

To trap Ingres errors locally, you must define an Ada error function and pass it to the EQUEL runtime routines for custom error management. The program error handler must be declared as an ADA function that can be exported. Because the Ada pragma **export_function** is used, the whole function must be in a package declared at the outermost scope.

The following format should be used to declare and define the function:

```
package Error_Trap is
 function Error_Proc( err: Integer ) return Integer;
 pragma export_function( Error_Proc );
end Error_Trap;

package body Error_Trap is
 function Error_Proc( err: Integer ) return Integer is
 begin
   ...
 end Error_Proc;
end Error_Trap;
```

This function must be passed to the EQUEL procedure **IIseterr** for runtime bookkeeping, using the Ada statement:

```
IIseterr( Error_Proc'Address );
```

The procedure **IIseterr** is declared externally for you by EQUEL.

This forces all runtime Ingres errors through your function, passing the Ingres error number as an argument. If you choose to handle the error locally and suppress Ingres error message printing the function should return 0; otherwise the function should return the Ingres error number received.

Avoid issuing any EQUEL statements in a user-written error handler defined to **IIseterr**, except for informative messages, such as **message, prompt, sleep** and **clear screen,** and messages that close down an application, such as **endforms** and **exit.**

The example below demonstrates a typical use of an error function to warn users of access to protected tables. This example passes through all other errors for default treatment.

```
package Error_Trap is
 function Error_Proc( ingerr: Integer ) return Integer;
 pragma export_function( Error_Proc );
end Error_Trap;

with text_io; use text_io;

package body Error_Trap is
 function Error_Proc( ingerr: Integer ) return Integer is
          -- Error number for protected tables
          TBLPROT: constant := 5003;
     begin
          if (ingerr = TBLPROT) then
              put_line( "No authorization for operation.");
              return 0; -- Suppress Ingres
                        -- printing message
          else
              return ingerr; -- Ingres will print message
          end if;
     end Error_Proc;
end Error_Trap;

-- In main procedure body issue the following statement
IIseterr( Error_Proc'Address );
```

Precompiling, Compiling and Linking an EQUEL Program

This section describes the EQUEL preprocessor for Ada, and the steps required to precompile, compile, and link an EQUEL program.

Generating an Executable Program

Once you have written your EQUEL program, it must be preprocessed to convert the EQUEL statements into Ada code. This section describes the use of the EQUEL preprocessor. Additionally, it describes how to compile and link the resulting code.

The EQUEL Preprocessor Command

The Ada preprocessor is invoked by the following command line:

eqa {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by EQUEL. The source file name, line number, and the erroneous statement itself are printed along with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If the **-f** flag is specified without a *filename*, the output is sent to standard output, one screen at a time. If the **-f** flag is omitted, output is given the basename of the input file, suffixed ".ada".
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename**.lis***, where *filename* is the name of the input file.
-lo	Like **-l**, but the generated Ada code also appears in the listing file.
-n. *ext*	Specifies the extension used for filenames in **## include** and ## include inline statements in the source code. If **-n** is omitted, include filenames in the source code must be given the extension ".qa".
-s	Reads input from standard input and generates Ada code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Ctrl Z**.
-w	Prints warning messages.
-?	Shows what command line options are available for **eqa**.

The EQUEL/Ada preprocessor assumes that input files are named with the extension ".qa". This default can be overridden by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated Ada statements with the same name and the extension ".ada".

If you enter the command without specifying any flags or a filename, Ingres displays a list of flags available for the command.

The following table presents the options available with **eqa**.

Eqa Command Examples

Command	Comment
eqa file1	Preprocesses "file1.qa" to "file1.ada"
eqa -l file2.xa	Preprocesses "file2.xa" to "file3.ada" and creates listing "file3.lis"
eqa -s	Accepts input from standard input and writes generated code to standard output
eqa -ffile4.out file4	Preprocesses "file4.qa" to "file4.out"
eqa	Displays a list of flags available for this command.

The ACS Environment and the Ada Compiler

The EQUEL/Ada preprocessor generates Ada code that you compile into your program library. You should use the VMS **ada** command to compile this code into your Ada program library.

The following sections describe the Ada program library and EQUEL programs.

Note: Check the Readme file for any operating system specific information on compiling and linking EQUEL/Ada programs.

Entering EQUEL Package Specifications

Once you have set up an Ada program library, you must add two EQUEL units to your library. The units are package specifications that describe to the Ada compiler all the calls that the preprocessor generates. The source for both these units is in the file:

ii_system:[ingres.files]eqdef.ada

Once you have defined your current program library by means of the **acs set library** command, you should enter the two units into your program library by issuing the following commands:

```
$ copy ii_system:[ingres.files]eqdef.ada []
$ ada eqdef.ada
$ delete eqdef.ada
```

The last step is not needed if you intend to compile the *closure* of a particular program from the source files at a later date. However, you should not modify the file if it is left in your directory.

The two EQUEL units need only be entered once into your program library. Of course, if a new release of EQUEL/Ada includes modifications to the file "eqdef.ada," the file should be copied and recompiled.

By issuing the following command, you will find the two new unit names "EQUEL" and "EQUEL_FORMS" in the library.

```
$ acs dir equel*
```

Defining Long Floating-point Storage

The storage representation format of long floating-point variables must be **d_float**. (For information, see Ada Variables and Data Types in this chapter.) This is because the EQUEL runtime system uses that format for floating-point conversions. If your EQUEL program has **long_float** variables that interact with the EQUEL runtime system, you must make sure they are stored in the **d_float** format. The default Ada format is **g_float**. A convenient way to control the format of all long float variables is to issue the **acs set pragma** program command. For example, by issuing the following command you redefine the program library characteristics for **long_float** from the default to **d_float**:

```
$ acs set pragma/long_float=d_float
```

A second remedy to this particular problem is to issue the statement:

```
pragma long_float(d_float)
```

in the source file of each compilation unit that uses floating-point variables. You may also explicitly declare the EQUEL variables with type **d_float**, as defined in package SYSTEM.

The following example is a typical command file that sets up a new Ada program library with the EQUEL package specifications and the **d_float** numerical format. The name of the new program library is passed in as a parameter:

```
$ acs create library [.'p1']
$ acs set library [.'p1']
$ acs set pragma/long_float=d_float
$ copy ii_system:[ingres.files]eqdef.ada []
$ ada eqdef.ada
$ delete eqdef.ada.
$ exit
```

The Ada Compiler

Once you have entered the EQUEL packages into the Ada program library, you can compile the Ada file generated by the preprocessor. The following example preprocesses and compiles the file "test1." Note that both the EQUEL/Ada preprocessor and the Ada compiler assume the default extensions:

```
$ eqa test1
$ ada/list test1
```

Linking an EQUEL Program

EQUEL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an EQUEL program, you can link it. Assuming your program unit is called "dbentry," use the following link command:

```
$ acs link dbentry,-
 ii_system:[ingres.files]equel/opt
```

It is recommended that you do not explicitly link in the libraries referenced in the EQUEL.OPT file. The members of these libraries change with different releases of Ingres. Consequently, you may be required to change your link command files in order to link your EQUEL programs.

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *QUEL Reference Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry") by listing it in the link command, as in the following example:

```
$ acs link formentry,-
 empform.obj,-
 ii_system:[ingres.files]equel/opt
```

Linking an EQUEL Program without Shared Libraries

While the use of shared libraries in linking EQUEL programs is recommended for optimal performance and ease of maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by EQUEL are listed in the equel.noshare options file. The options file must be included in your link command *after* all user modules. The libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an EQUEL program unit called "dbentry" that has been preprocessed and compiled:

```
$ acs link dbentry,-
 ii_system:[ingres.files]equel.noshare/opt
```

Include File Processing

The EQUEL/Ada **include** statement provides a means to include external packages and source files into your program's source code. The syntax of the statement is:

include *filename*

where *filename* is a quoted string constant specifying a file name or a logical name that points to the file name. If the file is in the local directory, it can also be specified without the surrounding quotes.

Including and Processing EQUEL/Ada Package Specifications

The above variant of the **include** statement can be used only to include package specifications. The preprocessor reads the specified file, processing all variables declared in the package, and generates the Ada **with** and **use** clauses using the last component of the file name (excluding the file extension) as the package name. If the last component of the file name has a trailing underscore, as is the standard in VAX/VMS Ada package specification files, then that trailing underscore is removed in the generated context clauses. The preprocessor does not generate an output file because it is assumed that the package specification has already been compiled.

The following example demonstrates this variant of the **include** statement. Assume that the specification of package "employee" is in file "employee_.qa" and that a procedure "empentry" is in file "empentry.qa":

Contents of "employee_.qa":

```
## package employee is
## ename:   String(1..20);
## eage:    Integer;
## esalary: Float;
## end employee;
```

Contents of "empentry.qa":

```
## include "[joe.ada.empfiles]employee_.qa"

## procedure empentry is
## begin
   -- Statements using variables in package "employee"
## end empentry;
```

The EQUEL/Ada preprocessor modifies the **include** line to the Ada **with** and **use** clauses by extracting the last component of the file name:

```
with employee;
use employee;
```

The above two clauses appear in the output file "empentry.ada." The preprocessor does not generate an output file for "employee_.qa," and the package "employee" must have already been compiled in order to compile the "empentry.ada" file.

Assuming that the files "employee_.qa" and "empentry.qa" appear as shown above, the following sequence of VMS commands should be executed in order to compile "empentry.ada":

```
$ eqa employee_.qa
$ eqa empentry.qa
$ ada employee_.ada
$ ada empentry.ada
```

You must still follow the Ada rules specifying the order of compilation. The EQUEL preprocessor does not affect these compilation rules.

Including EQUEL/Ada Source Code

In order to include source code into your EQUEL/Ada file, you should issue the EQUEL **include** statement with the **inline** option. Its syntax is as follows:

include inline *filename*

where *filename* has the same rules as mentioned earlier.

With this variant of **include**, the included text is preprocessed into the parent output file. For example, if you have a file called "messages.qa" that contains the text:

```
## message buffervar
## sleep 2
```

and you are preprocessing the file called "retrieve.qa", then the following **include** statement is legal in "retrieve.qa":

```
## retrieve (buffervar = e.name)
##
## include inline "messages.qa";
##
```

The file "messages.qa" is preprocessed into the output file "retrieve.ada." For more information on the **inline** option see the *QUEL Reference Guide.*

Coding Requirements for Writing EQUEL Programs

The following sections describe coding requirements for writing EQUEL programs.

Comments Embedded in Ada Output

Each EQUEL statement generates one comment and a few lines of ADA code. You may find that the preprocessor translates 50 lines of EQUEL into 200 lines of Ada. This may result in confusion about line numbers when you are debugging the original source code. To facilitate debugging, each group of Ada statements associated with a particular statement is preceded by a comment corresponding to the original EQUEL source. (Note that only *executable* EQUEL statements are preceded by a comment.) Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

Ada Blocks Generated by EQUEL

EQUEL statements that are associated with a block of code delimited by the braces { and }, or **begin** and **end**, are called block-structured statements. All the EQUEL block-structured statements generate Ada blocks. If there is no code contained in the block, EQUEL may need to generate the Ada **null** statement, depending on the type of Ada block generated. Consequently, if you *do* want an empty block, do not place just an Ada comment inside it (without the ## to delimit the comment), because the preprocessor would consider the comment to be Ada host code and would treat the block as a block containing Ada code.

For example, to disable the scrolling down of a table field, you might mistakenly code the following **activate** block:

```
## activate scrolldown employee -- this example contains
                                -- an error
##
   -- Disable scrolling of table field
##
```

The Ada comment in the block is considered Ada host code, and therefore, the **null** statement is not generated. This would later cause an Ada compiler syntax error. To resolve this situation, you must either let EQUEL know that the statement is only a comment, so that it will generate the **null** statement, or else code the **null** statement explicitly. The above example should be written as:

```
## activate scrolldown employee
##
## -- Disable scrolling of table field
##
```

or

```
## activate scrolldown employee
##
   -- Disable scrolling of table field
   null;
##
```

An EQUEL Statement that Does Not Generate Code

The **declare cursor** statement does not generate any Ada code. This statement should not be coded as the only statement in Ada constructs that does not allow *null* statements. For example, coding a **declare cursor** statement as the only statement in an Ada **if** statement not bounded by left and right braces would cause compiler errors:

```
if (using_database)
##   declare cursor empcsr for retrieve (employee.ename)
else
     put-line("You have not accessed the database");
end if;
```

The code generated by the preprocessor would be:

```
if (using_database)
else
     put_line("You have not accessed the database");
end if;
```

which is an illegal use of the Ada **if-then-else** statement.

EQUEL/Ada Preprocessor Errors

To correct most errors, you may wish to run the EQUEL preprocessor with the listing (**-l**) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to the Ada language, see the next section.

Preprocessor Error Messages

The following is a list of error messages specific to the Ada language.

E_E60001 "The ADA variable '%0c' is an array and must be subscripted."

Explanation: A variable declared as an array must be subscripted when referenced. The preprocessor does not confirm that you use the correct number of subscripts. A variable declared as a 1-dimensional array of characters, must not be subscripted as it refers to a character string.

E_E60002 "The ADA variable '%0c' is not an array and must not be subscripted."

Explanation: A variable not declared as an array cannot be subscripted. You cannot subscript string variables in order to refer to a single character or a slice of a string (substring).

E_E60003 "The ADA identifier '%0c' is not a declared type."

Explanation: The identifier was used as an Ada type name in an object or type declaration. This identifier has not yet been declared to the preprocessor and is not a preprocessor-predefined type name.

E_E60004 "The ADA CHARACTER variable '%0c' must be a 1-dimensional array."

Explanation: Variables of type CHARACTER can only be declared as 1-dimensional arrays. You cannot use a single character or a multidimensional array of characters as an Ingres string. Note that you can use a multidimensional array of type STRING.

E_E60005 "The ADA DIGITS clause '%0c' is out of the range 1..16."

Explanation: Embedded Ada supports D_FLOAT floating-point variables. Consequently, all DIGITS specifications must be in the specified range.

E_E60006 "Statement '%0c' is embedded in INCLUDE file package specification."

Explanation: Preprocessor INCLUDE files may only be used for Ada package specifications. The preprocessor generates an Ada WITH clause for the package. No executable statements may be included in the file because the code generated will not be accepted by the Ada compiler in a package specification.

E_E60007 "Too many names (%0c) in ADA identifier list. Maximum is %1c."

Explanation: Ada identifier lists cannot have too many names in the comma-separated name list. The name specified in the error message caused the overflow, and the remainder of the list is ignored. Rewrite the declaration so that there are fewer names in the list.

E_E60008 "The ADA identifier list has come up short."

Explanation: The stack used to store comma-separated names in Ada declarations has been corrupted. Try rearranging the list of names in the declaration.

E_E60009 "The ADA CONSTANT declaration of '%0c' must be initialized."

Explanation: CONSTANT declarations must include an initialization clause.

E_E6000A "The ADA identifier '%0c' is either a constant or an enumerated literal."

Explanation: The named identifier was used to retrieve data from Ingres. A constant, an enumerated literal and a formal parameter with the IN mode are all considered illegal for the purpose of retrieval.

E_E6000B "The ADA variable '%0c' with '.ALL' clause is illegal."

Explanation: The ADA .ALL clause, as specified with access objects, can be used only if the variable is an access object pointing at a single scalar-valued type. If the type is not scalar valued, or if the access object is pointing at a record or array, then the use of .ALL is illegal.

E_E6000C "The ADA variable '%0c' with '.ALL' clause is not a scalar type."

Explanation: The Ada .ALL clause, as specified with access objects, can be used only if the variable is an access object pointing at a single scalar-valued type. If the type is not scalar valued, or if the access object is pointing at a record or array, then the use of .ALL is illegal.

E_E6000D "Last component in ADA record qualification '%0c' is illegal."

Explanation: The last component referenced in a record qualification is not a member of the record. If this component was supposed to be declared as a record, the following components will cause preprocessor syntax errors.

E_E6000E "In ADA RENAMES statement, '%0c' must be a constant or a variable."

Explanation: The target object of a RENAMES statement must be a constant or a variable, and the item being declared is used a synonym for the target object.

E_E6000F "In ADA RENAMES statement, object is incompatible with type."

Explanation: The type of the target object in the RENAMES statement must be compatible in base type, size and array dimensions with the type name specified in the declaration.

E_E60010 "Only one name may be declared in an Ada RENAMES statement."

Explanation: One object can rename only one other object.

E_E60011 "Unclosed ADA block. There are %0c block(s) left open."

Explanation: If a file is terminated early or the END statement closing an Ada compilation unit is missing, this error will occur. If syntax errors were issued while parsing the compilation unit header, correct those errors first.

E_E60012 "The ADA variable '%0c' has not been declared."

Explanation: The named identifier was used where a variable must be used to set or retrieve Ingres data. The variable has not yet been declared.

E_E60013 "The ADA type %0c is not supported."

Explanation: Some Ada types are not supported because they are not compatible with the Ingres runtime system.

E_E60014 "The ADA variable '%0c' is a record, not a scalar value."

Explanation: The named variable qualification refers to a record. It was used where a variable must be used to set or retrieve Ingres data. This error may also cause syntax errors on record component references.

E_E60015 "You must issue a '## WITH %0c' before statement '%1c'."

Explanation: If your compilation unit includes forms statements you must issue the WITH EQUEL_FORMS clause. Otherwise you must issue the WITH EQUEL clause.

E_E60016 "The ADA statement %0c is not supported."

Explanation: Statements that modify the internal representation of variables that interact with Ingres are not supported.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Department information is stored in program variables. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo (the "toberesolved" database table, described below) to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second is for the Employee table. The **create** statements used to create the tables are shown below. The cursors retrieve all the information in their respective tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, both from the Department table and the Employee table, is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the EQUEL statements. The program illustrates table creation, multi-query transactions, all cursor statements and direct updates. For purposes of brevity, error handling on data manipulation statements is simply to close down the application.

The following two **create** statements describe the Employee and Department database tables:

```
##    create dept
##         (name      = c12,   -- Department name
##          totsales  = money, -- Total sales
##          employees = i2)    -- Number of employees

##    create employee
##        (name       = c20,   -- Employee name
##         age        = i1,    -- Employee age
##         idno       = i4,    -- Unique employee id
##         hired      = date,  -- Date of hire
##         dept       = c10,   -- Employee department
##         salary     = money) -- Yearly salary

-- Package:   Long_Float_Text_IO
-- Purpose:   Create text I/O package for LONG_FLOAT so as not to
--            conflict with the default G_FLOAT format. This
--            example assumes that the ACS SET PRAGMA command has
--            been issued for LONG_FLOAT.

with text_io;
package long_float_text_io is new text_io.float_io(long_float);

-- Package:   Trap_Error
-- Procedure: Close_Down
-- Purpose:   To trap Ingres runtime error messages. This
--            package defines the procedure Close_Down which is
--            called when a Ingres error is returned. The
--            procedure Close_Down is
--            passed to the runtime system via IIseterr.
--            When Close_Down is called, the error is printed
--            and the database session is terminated. Any open
--            transactions and cursors are implicitly closed.
-- Parameters:
--             ingerr - Integer containing Ingres
--             error number.

with text_io;               use text_io;
## with EQUEL;

##      package Trap_Error is

            function Close_Down(ingerr: Integer) return Integer;
            pragma export_function(Close_Down);
            ingres_error: Exception;

##      end Trap_Error;

##      package body Trap_Error is

##          function Close_Down(ingerr: Integer) return Integer is
##                error_text: String(1..200);
##          begin
##            inquire_ingres (error_text = errortext)
##            exit
              put_line("Closing down because of database error:");
              put_line(error_text);
              raise ingres_error;
              return ingerr;
##          end Close_Down;

##      end Trap_Error;
```

```
-- I/O utilities
with text_io;                          use text_io;
with integer_text_io;                  use integer_text_io;
with short_integer_text_io;            use short_integer_text_io;
with short_short_integer_text_io;      use short_short_integer_text_io;
with float_text_io;                    use float_text_io;
with long_float_text_io;               use long_float_text_io;
with trap_error;                       use trap_error;

## with EQUEL;

-- Procedure: Process_Expenses -- MAIN
-- Purpose:   Main body of the application. Initialize the
--            database, process each department, and terminate
--            the session.
-- Parameters:
--            None

##      procedure Process_Expenses is

              -- Function: Init_Db
              -- Purpose:  Initialize the database.
              --           Connect to the database, and abort on
              --           error. Before processing departments
              --           and employees create the table for
              --            employees who lose their department,
              --           "toberesolved".
              --            Initiate the multi-statement
              --           transaction.
              -- Parameters:
              --            None
              -- Returns:
              --            TRUE is initialized, FALSE if error.

##            function Init_Db return Boolean is

##                     create_err: Integer;

##            begin
##                    ingres personnel

                     put_line("Creating ""To_Be_Resolved"" table.");
##                     create toberesolved
##                              (name    = char(20),
##                               age     = integer1,
##                               idno    = integer4,
##                               hired   = date,
##                               dept    = char(10),
##                               salary  = money)

                       -- Was the create successful ?
##                     inquire_ingres (create_err = errorno)
                       if (create_err > 0) then
                        put_line("Fatal error on table creation.");
                           return FALSE;
                       else
                           -- Inform Ingres runtime system
                           -- about the errorhandler. All errors
                           -- from here on close down the
                           -- application.
                           IIseterr(Close_Down'Address);
##                         begin transaction
                           return TRUE;
                       end if;

##            end Init_Db;
```

```
                -- Procedure: End_Db
                -- Purpose:   Commit the multi-statement transaction
                --            and access to the database.
                -- Parameters:
                --            None

##              procedure End_Db is
##              begin
##                    end transaction
##                    exit
##              end _Db;

                -- Procedure:  Process_Employees
                -- Purpose:    Scan through all the employees for a
                --         particular department. Based on given
                --         conditions the employee may be or take a
                --         salary reduction.
                --          1.  If an employee was hired since
                --             1985 then the employee is terminated.
                --          2. If the employee's yearly salary is
                --             more than the minimum company wage of
                --             $14,000 and the employee is not close
                --             to retirement (over 58 years of age),
                --             then the employee takes a 5% salary
                --             reduction.
                --          3. If the employee's department is
                --             dissolved and the employee is not
                --             terminated, then the employee is
                --              moved into the "toberesolved" table.
                -- Parameters:
                --     dept_name    - Name of current department.
                --     deleted_dept - Is department dissolved?
                --     emps_term    - Set locally to record how many
                --                    employees were terminated
                --                     for the current department.

##              procedure Process_Employees
##                              (dept_name:      in String;
##                              deleted_dept:   in Boolean;
##                               emps_term:      in out Integer) is

##               salary_reduc: constant float = 0.95;
##               min_emp_salary: constant float := 14000.00;
##               nearly_retired: constant Short_Short_Integer := 58;

                -- Emp_Rec corresponds to the "employee" table
##              type Emp_Rec is
##                    record
##                              name:           String(1..20);
##                              age:            Short_Short_Integer;
##                                     idno:           Integer;
##                                     hired:          String(1..25);
##                                     salary:         Float;
##                                     hired_since_85: Integer;
##                              end record;
##                    emp: Emp_Rec;

##                  no_rows:   Integer;          -- Cursor control

                    title:     String(1..12);    -- Formatting values
                    descript:  String(1..25);

## begin
```

```
        -- Note the use of the Ingres function to find out
        -- who was hired since 1985.

##              range of e is employee

##              declare cursor empcsr for
##             retrieve (e.name, e.age, e.idno, e.hired, e.salary,
##                       res = int4(interval("days",
##                       e.hired-date("01-jan-1985"))))
##                  where e.dept = dept_name
##                  for direct update of (name, salary)

                 no_rows := 0;
                 emps_term := 0; -- Record how many

##               open cursor empcsr

                 while (no_rows = 0) loop

##              retrieve cursor empcsr (emp.name, emp.age, emp.idno,
##                                      emp.hired, emp.salary,
##                                      emp.hired_since_85)
##                 inquire_equel (no_rows = endquery)

                    if (no_rows = 0) then

                        -- Terminate if new employee
                        if (emp.hired_since_85 > 0) then

##                          delete cursor empcsr
                            title := "Terminated: ";
                            descript := "Reason: Hired since 1985.";
                             emps_term := emps_term + 1;

                       -- Reduce salary if large and not nearly retired
                            elsif (emp.salary > MIN_EMP_SALARY) then

                                 if (emp.age < NEARLY_RETIRED) then
##                                       replace cursor empcsr
##                                          (salary =
##                                           salary * SALARY_REDUC)
                                       title := "Reduction: ";
                                       descript :=
                                                 "Reason: Salary. ";
                                 else
                                       -- Do not reduce salary
                                       title := "No Changes: ";
                                     descript := "Reason: Retiring. ";
                                 end if;

                        -- Leave employee as is - low salary
                        else

                                     title = "No Changes: ";
                                     descript = "Reason: Salary. ";

                        end if;

                    -- Was employee's department dissolved ?
                    if (deleted_dept) then
##                           append to toberesolved (e.all)
##                                where e.idno = emp.idno
##                           delete cursor empcsr
                    end if;
```

```
                -- Log the employee's information
                put(" " & title & " ");
                put(emp.idno, 6);
                put(", & emp.name & ", ");
                put(emp.age, 3);
                put(", ");
                put(emp.salary, 8, 2, 0);
                put_line(" ; " & descript);

            end if; -- If a row was retrieved

        end loop;   -- Continue with cursor loop

##      close cursor empcsr

## end Process_Employees;

-- Procedure:   Process_Depts
-- Purpose:     Scan through all the departments, processing
--              each one. If the department has made less
--              than $50,000 in sales, then the department
--              is dissolved.
--              For each department process all the
--              employees (they may even be moved to another
--              database table).
--              If an employee was terminated, then update
--              the department's employee counter.
-- Parameters:
--              None

## procedure Process_Depts is

            MIN_TOT_SALES: constant := 50000.00;

            -- Dept_Rec corresponds to the "dept" table
##          type Dept_Rec is
##                record
##                       name: String(1..12);
##                       totsales: Long_Float;
##                       employees: Short_Integer;
##                end record;
##          dpt: Dept_Rec;

##       emps_term: Integer := 0;   -- Employees terminated
         deleted_dept: Boolean;      -- Was the dept deleted?
         dept_format: String(1..20); -- Formatting value
##       no_rows:     Integer;       -- Cursor control

##    begin
##        range of d is dept

##        declare cursor deptcsr for
##           retrieve (d.name, d.totsales, d.employees)
##           for direct update of (name, employees)
      no_rows := 0;
      emps_term := 0;
##    open cursor deptcsr

      while (no_rows = 0) loop
##          retrieve cursor deptcsr      (dpt.name,
##                                      dpt.totsales,
##                                      dpt.employees)
##          inquire_equel (no_rows = endquery)
            if (no_rows = 0) then
```

```
                -- Did the department reach minimum sales?
                if (dpt.totsales < MIN_TOT_SALES) then
##                      delete cursor deptcsr
                        deleted_dept := TRUE;
                        dept_format := " -- DISSOLVED --";
                else
                        deleted_dept := FALSE;
                        dept_format := (1..20 => ' ');
                end if;

                -- Log what we have just done
                put("Department: " & dpt.name &
                    ", Total Sales: ");
                put(dpt.totsales, 12, 3, 0);
                put_line(dept_format);

                -- Now process each employee in the department
                Process_Employees(dpt.name,
                                  deleted_dept,
                                  emps_term);

                -- If employees were terminated, record it
                if (emps_term > 0 and not deleted_dept) then
##                       replace cursor deptcsr
##                       (employees = employees - emps_term)
                end if;

            end if;          -- If a row was retrieved

        end loop;            -- Continue with cursor loop

## close cursor deptcsr

## end Process_Depts;

## begin        -- MAIN program

            put_line("Entering application to process expenses.");
              if (Init_Db) then
                   Process_Depts;
                   End_Db;
              end if;
              put_line("Completion of application.");

              exception

                when ingres_error => -- Raised by Close_Down
                 put_line("Contact your database administrator.");

## end Process_Expenses;
```

The Employee Query Interactive Forms Application

This EQUEL/FORMS application uses a form in **query** mode to view a subset of the Employee table in the Personnel database. An Ingres query qualification is built at runtime using values entered in fields of the form "empform."

The objects used in this application are:

Object	Description
personnel	The program's database environment.
employee	A table in the database, with six columns: name (**c20**) age (**i1**) idno (**i4**) hired (**date**) dept (**c10**) salary (**money**).
empform	A VIFRED form with fields corresponding in name and type to the columns in the Employee database table. The name and idno fields are used to build the query and are the only updatable fields. "Empform" is a compiled form.

The application is driven by a **display** statement that allows the runtime user to enter values in the two fields that will build the query. The Build_Query and Exec_Query procedures make up the core of the query that is run as a result. Note the way the values of the query operators determine the logic used to build the **where** clause in Build_Query. The **retrieve** statement encloses a **submenu** block that allows the user to step through the results of the query.

No updates are performed on the values retrieved, but any particular employee screen may be saved in a log file through the **printscreen** statement.

The following **create** statement describes the format of the Employee database table:

```
## create employee
##         (name    = c20,   -- Employee name
##          age     = i1,    -- Employee age
##          idno    = i4,    -- Unique employee id
##          hired   = date,  -- Date of hire
##          dept    = c10,   -- Employee department
##          salary  = money) -- Annual salary

##    package Compiled_Empform is
##              empform: Integer;
##              pragma import_object( empform );
##    end Compiled_Empform;
```

```
      with Compiled_Empform; use Compiled_Empform;
      with Text_Io;  use Text_Io;
      with Integer_Text_Io; use Integer_Text_Io;
##    with equel_forms;

##    procedure Employee_Query is

      -- Initialize global WHERE clause qualification buffer to
      -- be an Ingres default qualification that is
      -- always true.
##    where_clause: String(1..100) :=
##     ('1', '=', '1', others => ' ');

        -- Procedure: Build_Query
        -- Purpose:   Build an Ingres query from the values in the "name" and
        --            "idno" fields in "empform".
        -- Parameters:
        --             None

##      procedure Build_Query is

##            ename: String(1..20);
##            eidno: Integer;

              -- Query operator table that maps integer values to
              -- string query operators.
##            operators: array(1..6) of String(1..2) :=
##                    ("= ", "!", "< ", "> ", "<=", ">=");

              -- Operators corresponding to the two fields,
              -- that index into the "operators" table.
##            opername, operidno: Integer;

##      begin
##            getform #empform
##                (ename = name, opername = getoper(name),
##                 eidno = idno, operidno = getoper(idno))

              -- Fill in the WHERE clause
              where_clause := (1..100 => ' ');
              if (opername = 0 and operidno = 0) then

                  -- Default qualification
                  where_clause(1..3) := "1=1";

              elsif (opername = 0 and operidno /= 0) then

                  -- Query on the "idno" field
                  where_clause(1..8) :=
                        "e.idno" & operators(operidno);
                  put( where_clause(9..100), eidno );
              elsif (opername /= 0 and operidno = 0) then

                  -- Query on the "name" field
                  where_clause(1..30) :=
                      "e.name" & operators(opername) &
                      """" & ename & """";

              else -- (opername /= 0 and operidno /= 0)

                  -- Query on both fields
                  where_clause(1..43) :=
                       "e.name" & operators(opername) &
                       """" & ename & """ and " &
                       "e.idno" & operators(operidno);
                  put( where_clause(44..100), eidno );
```

```
            end if;

##    end Build_Query;

      --    Procedure: Exec_Query
      --    Purpose:   Given a query buffer, defining a WHER
      --               clause issue a RETRIEVE to allow the
      --               runtime use to
      --               browse the employees found with the given
      --               qualification.
      -- Parameters:
      --             None

##    procedure Exec_Query is

##          ename:   String(1..20); -- Employee data
##          eage:    Short_Integer;
##          eidno:   Integer;
##          ehired:  String(1..25);
##          edept:   String(1..10);
##          esalary: Float;

            rows: Boolean := FALSE; -- Were rows found
##    begin
          -- Issue query using WHERE clause
##        retrieve (
##                   ename = e.name, eage = e.age,
##                   eidno = e.idno, ehired = e.hired,
##                   edept = e.dept, esalary = e.salary)
##        where where_clause
##        {
              rows := TRUE;

              -- Put values up and display them
##            putform #empform (
##            name = ename, age = eage,
##            idno = eidno, hired = ehired,
##            dept = edept, salary = esalary)
##            redisplay

##            submenu
##            activate menuitem "next", frskey4
##            {
                   -- Do nothing, and continue with the
                   -- RETRIEVE loop. The last one will
                   -- drop out.
                   null;
##            }
##            activate menuitem "Save", frskey8
##            {
                   -- Save screen data in log file
##                 printscreen (file = "query.log")
```

```
                    -- Drop through to next employee
##            }
##            activate menuitem "End", frskey3
##            {
                    -- Terminate the RETRIEVE loop
##                  endretrieve
##            }
##      }
        if (not rows) then
##          message "No rows found for this query"
        else
##          clear field all
##          message "Reset for next query"
        end if;

##      sleep 2

##    end Exec_Query;
## begin

##      forms
##      message "Accessing Employee Query Application . . ."
##      ingres personnel

##      range of e is employee

##      addform empform

##      display #empform query
##      initialize

##      activate menuitem "Reset"
##      {
##            clear field all
##      }

##      activate menuitem "Query"
##      {
                    -- Verify validity of data
##                  validate

                    Build_Query;
                    Exec_Query;
##      }

##      activate menuitem "LastQuery"
##      {
             Exec_Query;
##      }

##      activate menuitem "End"
##      {
##           breakdisplay
##      }
##      finalize

##      clear screen
##      endforms
##   exit

## end Employee_Query;
```

The Table Editor Table Field Application

This EQUEL/FORMS application uses a table field to edit the Person table in the Personnel database. It allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate their use and their interaction with an Ingres database.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**c20**) age (**i2**) number (**i4**) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**c20**) age (**i4**). When initialized, the table field includes the hidden number (**i4**) column.

At the start of the application, a **retrieve** statement is issued to load the table field with data from the Person table. Once the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated or deleted. When finished, the values are unloaded from the table field, and, in a multi-statement transaction, the user's updates are transferred back into the Person table.

The following **create** statement describes the format of the Person database table:

```
##   create person
##      (name = c20,  -- Person name
##       age = i2,    -- Age
##       number = i4) -- Unique id number

##    with equel_forms;

##    procedure Table_Edit is

        -- Person information corresponds to "person" table
##      pname:   String(1..20); -- Full name
##      page:    Short_Integer; -- Age
##      pnumber: Integer;       -- Unique person number
##      pmaxid:  Integer;       -- Maximum person id number
```

```
        -- Table field row states
        ROW_UNDEF:    constant := 0;-- Empty or undefined row
        ROW_NEW:      constant := 1;-- Appended by user
        ROW_UNCHANGE: constant := 2;
                                  -- Loaded by program - not updated
        ROW_CHANGE:   constant := 3;
                                   -- Loaded by program and updated
        ROW_DELETE:   constant := 4;-- Deleted by program

        -- Table field entry information
##      state,            -- State of data set row (see above)
##      recnum,           -- Record number
##      lastrow: Integer; -- Last row in table field

        -- Utility buffers
##      search:   String(1..20); -- Name to find in search loop
##      msgbuf:   String(1..80); -- Message buffer
##      password: String(1..13); -- Password buffer
##      respbuf:  String(1..1);  -- Response buffer

        -- Error handling variables for database updates
##      upd_err,                    -- Updates error
##      upd_rows:     Integer;      -- Number of rows updated
        upd_commit:   Boolean;      -- Commit updates
##      save_changes: Boolean;      -- Save changes or quit
##    begin
          -- Start up Ingres and the FORMS system
          -- We assume no Ingres errors will happen during
          -- screen updating

##        ingres "personnel"

##        forms

          -- Verify that the user can edit the "person" table
##        prompt noecho ("Password for table editor: ", password)
          if (password /= "MASTER_OF_ALL") then
##             message "No permission for task. Exiting . . ."
##             endforms
##             exit
               return;
          end if;

##        message "Initializing Person Form . . ."
##        forminit personfrm

           -- Initialize "persontbl" table field with a data set
           -- in FILL mode so that the runtime user can append
           -- rows. To keep track of events occurring to original
           -- rows that will be loaded into the table field, hide
           -- the unique person number.

##         inittable personfrm persontbl FILL (number = integer4)

           -- Load the information from the "person" table into
           -- the person variables. Also save away the maximum
           -- person id number.

##         message "Loading Person Information . . ."

##         range of p is person
```

```
            -- Fetch data into person record, and load table field
##          retrieve (pname = p.name, page = p.age,
##                    pnumber = p.number)
##          {
##                loadtable personfrm persontbl
##                   (name = pname, age = page, number = pnumber)
##          }

            -- Fetch the maximum person id number for later use.
            -- Performance note: max() will do sequential scan of
            -- table.
##          retrieve (pmaxid = max(p.number))

            -- Display the form and allow runtime editing
##          display personfrm update
##          initialize

##          -- Provide a menu, as well as the system FRS key to
##          -- scroll to both extremes of the table field. Note
##          -- that a comment between
##          -- DISPLAY loop components MUST be marked with a ##.

##          activate menuitem "Top", frskey5
##          {
##               scroll personfrm persontbl TO 1 -- Backward
##          }

##          activate menuitem "Bottom", frskey6
##          {
##               scroll personfrm persontbl to end -- Forward
##          }

##          activate menuitem "Remove"
##          {
                 -- Remove the person in the row the user's cursor
                 -- is on. If there are no persons, exit operation
                 -- with message. Note that this check cannot
                 -- really happen as there is always at least one
                 -- UNDEFINED row in FILL mode.

##               inquire_frs table personfrm
##                     (lastrow = lastrow(persontbl))
                 if (lastrow = 0) then
##                     message "Nobody to Remove"
##                     sleep 2
##                     resume field persontbl
                 end if;

##               deleterow personfrm persontbl
                               -- Recorded for later
##          }

##          activate menuitem "Find", frskey7
##          {
                 -- Scroll user to the requested table field
                 -- entry. Prompt the user for a name, and if one
                 -- is typed in loop through the data set
                 -- searching for it.

                 search := (1..20 = ' ');
##               prompt ("Person's name : ", search)
                 if (search(1) = ' ') then
##                     resume field persontbl
                 end if;
```

```
##              unloadtable personfrm persontbl
##                    (pname = name, recnum = _record,
##                     state = _state)
##              {
                     -- Do not compare with deleted rows
                     if (state /= row_delete and pname = search)
                         then
##                           scroll personfrm persontbl to recnum
##                           resume field persontbl
                     end if;
##              }

                -- Fell out of loop without finding name.
                -- Issue error.
                msgbuf := (1..80 = ' ');
                msgbuf(1..62) := "Person '" & search &
                    "' not found in table [HIT RETURN] ";
##              prompt noecho (msgbuf, respbuf)
##        }

##        activate menuitem "Save", frskey8
##        {
##              validate field persontbl
                save_changes := TRUE;
##              breakdisplay
##        }

##        activate menuitem "Quit", frskey2
##        {
                save_changes := FALSE;
##              breakdisplay
##        }
##        finalize

          if (save_changes) then

             -- Exit person table editor and unload the table
             -- field.If any updates, deletions or additions were
             -- made, duplicate these changes in the source
             -- table. If the user added new people we must
             -- assign a unique person
             -- id before returning it to the database table. To
             -- do this, we increment the previously saved
             -- maximum id number with each APPEND.

##           message "Exiting Person Application . . ."

             -- Do all the updates in a multi-statement
             -- transaction. For simplicity, this transaction does
             -- not restart on deadlock.

##           begin transaction
             upd_commit := TRUE;

             -- Handle errors in the UNLOADTABLE loop, as we
             -- want to cleanly exit the loop, after cleaning up
             -- the transaction.

##           unloadtable personfrm persontbl
##             (pname = name, page = age,
##              pnumber = number, state = _state)
##           {

               case (state) is

                    when row_new =>
```

```
                     -- Filled by user. Insert with new unique id
                        pmaxid := pmaxid + 1;
##                      repeat append to person
##                         (name = @pname,
##                          age = @page,
##                          number = @pmaxid);

                    when row_change =>

                        -- Updated by user. Reflect in table
##                      repeat replace p
##                         (name = @pname, age = @page)
##                         where p.number = @pnumber

                    when row_delete =>

                        -- Deleted by user, so delete from table.
                        -- Note that only original rows are saved
                        -- by the program, and not rows appended
                        -- at runtime.
##                      repeat delete p where p.number = @pnumber

                    when others =>

                        -- Else UNDEFINED or UNCHANGED
                        -- No updates required.
                        null;

                  end case;

                -- Handle error conditions -
                -- If an error occurred, then abort the
                -- transaction. If a no rows were updated then
                -- inform user, and prompt for continuation.

##            inquire_equel (upd_err = errorno, upd_rows = rowcount)

                if (upd_err > 0) then -- Abort on error

                        upd_commit := FALSE;
##                      message "Aborting updates . . ."
##                      abort
##                      endloop

                elsif (upd_rows = 0) then  -- May want to stop
                        msgbuf := (1..80 = ' ');
                        msgbuf(1..62) :=
                                  "Person '" & pname &
                               "' not updated. Abort all updates? ";
##                      prompt noecho (msgbuf, respbuf)
                        if (respbuf = "Y" or respbuf = "y") then
                              upd_commit := FALSE;
##                            abort
##                            endloop
                        end if;
                end if;
##          }

                if (upd_commit) then
##                  end transaction -- Commit the updates
                end if;

            end if;  -- If saving changes
```

```
##      endforms          -- Terminate the FORMS and Ingres
##      exit

## end Table_Edit;
```

The Professor-Student Mixed Form Application

This EQUEL/FORMS application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**c25**) pdept (**c10**) See its **create** statement below for a full description.
student	A database table with seven columns: sname (**c25**) sage (**i1**) sbdate (**c25**) sgpa (**f4**) sidno (**i4**) scomment (text(**200**)) sadvisor (**c25**) See the **create** statement below for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has the pname and pdept fields, which correspond to the information in the Professor table, and studenttbl table field. The pdept field is display-only. "Masterfrm" is a compiled form.
studenttbl	A table field in "masterfrm" with two columns, sname and sage. When initialized, it also has five more hidden columns corresponding to information in the Student table.

Object	Description
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the sgpa, scomment and sadvisor fields are updatable. All other fields are display-only. "Studentfrm" is a compiled form.
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the "studentfrm" form and the studenttbl table field.

The program uses the "masterfrm" as the general-level master entry, in which data can only be retrieved and browsed, and the "studentfrm" as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the pname (professor name) field and then selects the **Students** menu operation. The operation fills the displayed and hidden columns of the table field "studenttbl" with detailed information of the students reporting to the named professor.

The user may then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student may be requested by selecting the **Zoom** menu operation. This operation displays the form "studentfrm." The fields of "studentfrm" are filled with values stored in the hidden columns of "studenttbl."

The user may make changes to three fields (sgpa, scomment and sadvisor). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

The following two **create** statements describe the Professor and Student database tables:

```
##    create student       -- Graduate student table
##        (sname    = c25, -- Name
##         sage     = i1,  -- Age
##         sbdate   = c25, -- Birth date
##         sgpa     = f4,  -- Grade point average
##         sidno    = i4,  -- Unique student number
##         scomment = text(200), -- General comments
##         sadvisor = c25) -- Advisor's name

##    create professor  -- Professor table
##        (pname = c25, -- Professor's name
##         pdept = c10) -- Department

        -- Master and student compiled forms (imported objects)
##      package Compiled_Forms is
##              masterfrm, studentfrm: Integer;
                pragma import_object( masterfrm );
                pragma import_object( studentfrm );
##      end Compiled_Forms;
```

```
        with Compiled_Forms; use Compiled_Forms;
        with Text_Io;     use Text_Io;
        with Integer_Text_Io; use Integer_Text_Io;
##      with equel_forms;

        -- Procedure: Prof_Student
        -- Purpose:   Main body of "Professor Student"
        --             Master-Detail application.

##      procedure Prof_Student is

      -- Graduate student record maps to "student" database table
##       type Student_Rec is
##             record
##                     sname:    String(1..25);
##                     sage:     Short_Short_Integer;
##                     sbdate:   String(1..25);
##                     sgpa:     Float;
##                     sidno:    Integer;
##                     scomment: String(1..200);
##                     sadvisor: String(1..25);
##             end record;
##       grad: Student_Rec;

        -- Professor record maps to "professor" database table
##      type Prof_Rec is
##             record
##                     pname: String(1..25);
##                     pdept: String(1..10);
##             end record;
##      prof: Prof_Rec;

        -- Useful forms runtime information
##      lastrow,             -- Lastrow in table field
##      istable: Integer;    -- Is a table field?

         -- Utility buffers
##       msgbuf:  String(1..100); -- Message buffer
##       respbuf: String(1..1);   -- Response buffer
##       oldadv:  String(1..25);  -- Old advisor name before Zoom

         -- Function: Student_Info_Changed
         -- Purpose:  Allow the user to zoom into the details of
         --           a selected student. Some of the data can be
         --            updated by the user. If any updates were
         --           made, then reflect these back into the
         --           database table.The procedure returns TRUE if
         --          any changes were made.
         -- Parameters:
         --            None
         -- Returns:
         --            TRUE/FALSE - Changes were made to the
         --            database. Sets the global "grad" record
         --            with the new data.

##       function Student_Info_Changed return Boolean is

##             changed: Integer; -- Changes made to the form?
##             valid_advisor: Integer; -- Is advisor a professor?
##       begin
```

```
                -- Display the detailed student information
##              display #studentfrm update
##              initialize
                     (sname = grad.sname,
##                    sage = grad.sage,
##                    sbdate = grad.sbdate,
##                    sgpa = grad.sgpa,
##                    sidno = grad.sidno,
##                    scomment = grad.scomment,
##                    sadvisor = grad.sadvisor)

##              activate menuitem "Write"
##              {

                      -- If changes were made then update the
                      -- database table. Only bother with the
                      -- fields that are not read-only.

##                    inquire_frs form (changed = change)

                      if (changed = 1) then
##                        validate
##                        message "Writing to database. . ."

##                        getform
##                             (grad.sgpa = sgpa,
##                              grad.scomment = scomment,
##                              grad.sadvisor = sadvisor)

                          -- Enforce integrity of name
##                        retrieve (valid_advisor =
##                             count(p.pname
##                             where p.pname = grad.sadvisor))

                          if (valid_advisor = 0) then
##                                message "Not a valid name"
##                                sleep 2
##                                resume field sadvisor
                          end if;

##                        replace s
##                                (sgpa = grad.sgpa,
##                                 scomment = grad.scomment,
##                                 sadvisor = grad.sadvisor)
##                        where s.sidno = grad.sidno
                      end if;
##                    breakdisplay
##              }               -- "Write"

##              activate menuitem "End", frskey3
##              {
                        -- End without submitting changes
                        changed := 0;
##                      breakdisplay
##              }     -- "End"
##              finalize

                return (changed = 1);

##          end Student_Info_Changed;

##      begin

                -- Start up Ingres and the FORMS system
##              forms
```

```
##          message "Initializing Student Administrator . . ."
##          ingres personnel

##          range of p is professor, s is student

##          addform masterfrm
##          addform studentfrm

            -- Initialize "studenttbl" with a data set in READ
            -- mode. Declare hidden columns for all the extra
            -- fields that the program will display when more
            -- information is requested about a student. Columns
            -- "sname" and "sage" are displayed, all other columns
            -- are hidden, to be used in the student information form.

##          inittable #masterfrm studenttbl read
##                (sbdate = char(25),
##                 sgpa = float4,
##                sidno = integer4,
##                scomment = char(200),
##                sadvisor = char(20))

            -- Drive the application, by running "masterfrm", and
            -- allowing the user to "zoom" into a selected
            -- student.
##          display #masterfrm update

##          initialize
##          {
##                message "Enter an Advisor name . . ."
##                sleep 2
##          }

##          activate menuitem "Students", field "pname"
##          {
                  -- Load the students of the specified professor
##                getform (prof.pname = pname)

                  -- If no professor name is given then resume
                  if (prof.pname(1) = ' ') then
##                     resume field pname
                  end if;

                  -- Verify that the professor exists. If not
                  -- print a message, and continue. We assume
                  -- that each professor has exactly one department.
                  prof.pdept := (1..10 = ' ');
##                retrieve (prof.pdept = p.pdept)
##                     where p.pname = prof.pname

                  -- If no professor report error
                  if (prof.pdept(1) = ' ') then
                        msgbuf := (1..100 => ' ');
                        msgbuf(1..59) :=
                             "No professor with name '" &
                             prof.pname & "' [RETURN]";
##                      prompt noecho (msgbuf, respbuf)
##                      clear field all
##                      resume field pname
                  end if;

                  -- Fill the department field and load students
##                message "Retrieving Student Information . . ."
```

```
##                putform (pdept = prof.pdept)
##                clear field studenttbl
##                redisplay -- Refresh for query

                   -- With the advisor name, load into the
                   -- "studenttbl" table field all the graduate
                   -- students who report to the professor with that name.
                   -- Columns "sname" and "sage" will be displayed,
                   -- and all other columns will be hidden.

##             retrieve
##                   (grad.sname = s.sname,
##                    grad.sage = s.sage,
##                    grad.sbdate = s.sbdate,
##                    grad.sgpa = s.sgpa,
##                    grad.sidno = s.sidno,
##                    grad.scomment = s.scomment,
##                    grad.sadvisor = s.sadvisor)
##             where s.sadvisor = prof.pname
##             {
##                    loadtable #masterfrm studenttbl
##                           (sname = grad.sname,
##                            sage = grad.sage,
##                            sbdate = grad.sbdate,
##                            sgpa = grad.sgpa,
##                            sidno = grad.sidno,
##                            scomment = grad.scomment,
##                            sadvisor = grad.sadvisor)
##             }

##             resume field studenttbl
##      }       -- "Students"

##      activate menuitem "Zoom"
##      {
                -- Confirm that user is on "studenttbl", and that
                -- the table field is not empty. Collect data
                -- from the row and zoom for browsing and updating.

##              inquire_frs field #masterfrm (istable = table)
                if (istable = 0) then
##                   prompt noecho
##                      ("Select from the student table [return]",
##                       respbuf)
##                   resume field studenttbl
                end if;

##              inquire_frs table #masterfrm (lastrow = lastrow)
                if (lastrow = 0) then
##                  prompt noecho
##                        ("There are no students [return]",
##                          respbuf)
##                  resume field pname
                end if;

                -- Collect all data on student into graduate
                -- record
##              getrow #masterfrm studenttbl
##                    (grad.sname = sname,
##                     grad.sage = sage,
##                     grad.sbdate = sbdate,
##                     grad.sgpa = sgpa,
##                     grad.sidno = sidno,
##                     grad.scomment = scomment,
##                     grad.sadvisor = sadvisor)
```

```
                oldadv := grad.sadvisor;

                -- Display "studentfrm", and if any changes were
                -- made make the updates to the local table field
                -- row. Only make updates to the columns
                -- corresponding to
                -- writable fields in "studentfrm". If the student
                -- changed advisors then delete this row from display.

                if (Student_Info_Changed) then
                      if (grad.sadvisor /= oldadv) then
##                         deleterow #masterfrm studenttbl
                      else
##                         putrow #masterfrm studenttbl
##                                  (sgpa = grad.sgpa,
##                                   scomment = grad.scomment,
##                                   sadvisor = grad.sadvisor)
                      end if;
                end if;

##      }       -- "Zoom"

##      activate menuitem "Quit", frskey2
##      {
##           breakdisplay
##      }       -- "Quit"

##      finalize

##      clear screen
##      endforms
##      exit

## end Prof_Student;
```

Chapter 6: Embedded QUEL for BASIC

This chapter describes the use of EQUEL with the BASIC programming language.

Note: EQUEL/BASIC is supported in the VMS operating environment only.

EQUEL Statement Syntax for BASIC

This section describes the language-specific ground rules for embedding QUEL database and forms statements in a BASIC program. An EQUEL statement has the following general syntax:

*EQUEL_statement*

For information on QUEL statements, see the *QUEL Reference Guide*. For information on EQUEL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe how to use the various syntactical elements of EQUEL statements as implemented in BASIC.

BASIC Line Numbers and the EQUEL Mark

The BASIC line number, while not required, can occur at the beginning of most EQUEL statements before the EQUEL mark, ##. For example:

```
100  ##   destroy emp
```

The BASIC line number, if present, must be the first item on the line. It can be preceded only by spaces or tabs. The EQUEL ## mark must be the next item on the line after the optional line number and can be preceded by spaces or tabs. Any lines not marked by ## are treated as BASIC host code and are not preprocessed. Comments on lines not beginning with the ## mark are considered BASIC host code.

In most instances the preprocessor outputs any BASIC line numbers that precede an EQUEL statement. However, in a few cases the preprocessor ignores a BASIC line number and does not include it in the generated code. For example, line numbers occurring before EQUEL statements that produce no BASIC code are ignored by the preprocessor. Line numbers preceding EQUEL block statements, such as ## { and ## }, are also ignored. Line numbers should not occur on those lines containing a continued EQUEL string literal.

The preprocessor never generates line numbers of its own. Thus, if you prefix an EQUEL statement with a line number and that statement is translated by the preprocessor into several BASIC statements, the line number will appear before the first BASIC statement only. Subsequent BASIC statements will be unnumbered.

Note that the BASIC language does require a line number on the first line of a program or subprogram. The EQUEL preprocessor does not verify that these line numbers exist.

Terminator

No statement terminator is required for EQUEL/BASIC statements. It is conventional not to use a statement terminator in EQUEL statements, although the semicolon is allowed at the end of EQUEL statements. The preprocessor ignores it. For example, the following two statements are equivalent:

```
##   sleep 1
```

and

```
##   sleep 1;
```

The terminating semicolon may be convenient when entering code directly from the terminal using the **-s** flag. For information on using the **-s** flag to test the syntax of a particular EQUEL statement, see Precompiling, Compiling and Linking an EQUEL Program in this chapter.

EQUEL statements that are made up of a few other statements, such as a **display** loop, only allow a semicolon after the last statement. For example:

```
##  display empform             ! no semicolon here
##  initialize                  ! no semicolon here
##  activate menuitem "help"    ! no semicolon here
##  {
##      message "No help yet";  ! semicolon allowed
##      sleep 2;                ! semicolon allowed
##  }
##  finalize;                   ! Semicolon allowed on last statement
```

Variable declarations made visible to EQUEL observe the normal BASIC declaration syntax. Thus, variable declarations should not be terminated with a semicolon.

Line Continuation

There are no special line-continuation rules for EQUEL/BASIC. EQUEL statements can be broken between words and continued on any number of subsequent lines. An exception to this rule is that you cannot continue a statement between two words that are reserved when they appear together, such as **declare cursor**. For a list of double keywords, see the *QUEL Reference Guide*. Each continuation line must be started with ## characters. Blank lines are permitted between continuation lines. The BASIC line continuation symbol (**&**), cannot be used with EQUEL lines.

If you want to continue a character-string constant across two lines, end the first line with a backslash character (\), and continue the string at the beginning of the next line. In this case, do not place ## characters at the beginning of the continuation lines.

For examples of string continuation, see String Literals in this chapter.

The BASIC code generated by the EQUEL preprocessor will follow the BASIC rules for continuing statements. Host code should, of course, follow the BASIC rules for line continuation. For example:

```
##  message
##           "This is a message"
    print                          &
           "This is a message"
```

Comments

You can include a comment field or line in an EQUEL statement by typing the exclamation point (!) at the beginning of the comment field. The following example shows the use of a comment field on the same line as an EQUEL statement.

```
##  open cursor emp      ! Process employees
```

The next example shows the use of a comment field embedded in an EQUEL statement:

```
##  retrieve (namevar=e.ename)
##  ! confirm that "eno" was chosen
##  where e.eno = currentval
```

In both cases, the preprocessor ignores the comment field. Note that a comment field terminates with the new line. A comment field cannot be continued over multiple lines.

A comment line can appear anywhere in an EQUEL program that a blank line is allowed, with the following exceptions:

- In string constants. Such a comment would be interpreted as part of the string constant.

- Between two words that are reserved when they appear together, such as **declare cursor**. See the list of reserved words in the *QUEL Reference Guide*.
- In parts of statements that are dynamically defined. For example, a comment in a string variable specifying a form name is interpreted as part of the form name.

The following restrictions apply to BASIC comments that are not in lines beginning with ##:

- BASIC comments cannot appear between component lines of EQUEL block-type statements. These include **retrieve**, **initialize**, **activate**, **unloadtable**, **formdata**, and **tabledata**, all of which at least optionally have accompanying blocks delimited by open and close braces. BASIC comment lines must not appear between the statement and its block-opening delimiter.

  For example:

```
##   retrieve (ename = employee.name)
        ! Illegal to put a host comment here!
## {
        ! A host comment is perfectly legal here
        print "Employee name";ename
## }
```

- BASIC comments cannot appear between the components of compound statements, in particular the **display** statement. It is illegal for a BASIC comment to appear between any two adjacent components of the **display** statement, including **display** itself and its accompanying **initialize**, **activate**, and **finalize** statements.

  For example:

```
## display empform
         ! illegal to put a host comment here!
## initialize (empname = "frisco mcmullen")
         ! host comment illegal here!
## activate menuitem "clear"
## {
         ! host comment here is fine
##          clear field all
## }
         ! host comment illegal here!
## activate menuitem "end"
## {
##          breakdisplay
## }
         ! host comment illegal here!
## finalize
```

The *QUEL Reference Guide* specifies these restrictions on a statement-by-statement basis.

When the QUEL comment is delimited by /* and */ or appears on lines that begin with ##, it can be considered a valid EQUEL/BASIC comment and can span multiple lines.

String Literals

You can use either double quotes or single quotes to delimit string literals in EQUEL/BASIC, as long as the same delimiter is used at the beginning and the end of any one string literal.

To embed a double quote with a string literal, use single quotes as the string delimiter. Single quotes cannot be embedded in a string literal. If you want to embed single quotes in a character string, assign the string to a variable and use the variable in the EQUEL statement.

When continuing an EQUEL statement to another line in the middle of a string literal, use a backslash immediately prior to the end of the first line. In this case, the backslash and the following newline character are ignored by the preprocessor, so that the following line can continue both the string and any further components of the EQUEL statement. Any leading spaces on the next line are considered part of the string. For example, the following are legal EQUEL statements:

```
##  message 'Please correct errors found in updating \
the database tables.'

##  append to employee (empname = "Freddie \
Mac", empnum = 222)
```

Integer Literals

You can use the optional trailing percent sign (%) with EQUEL integer literals. The preprocessor always adds the percent sign to the integer literals that it generates.

BASIC Variables and Data Types

This section describes how to declare and use BASIC program variables in EQUEL.

Variable and Type Declarations

The following sections describe variable and type declarations.

EQUEL Variable Declarations Procedures

EQUEL statements use BASIC variables to transfer data from a database or a form into the program and conversely. You must declare BASIC variables to EQUEL before using them in EQUEL statements. The preprocessor does not allow implicit variable declarations. For this reason, the "%" and "$" suffixes cannot be used with variable names. BASIC variables are declared to EQUEL by preceding the declaration with the ## mark. The declaration must be in a position syntactically correct for the BASIC language.

In general, EQUEL variables can be referred to in the program or subprogram in which they are declared. The scope of variables is discussed in detail in a later section.

The Declare Ingres Statement

Prior to any EQUEL declarations or statements in your main program, you must issue the following statement:

```
## declare ingres
```

This statement causes the preprocessor to generate code to include a file of declarations needed by EQUEL at runtime. You will not be able to successfully link an EQUEL program without this statement. The statement also serves to terminate the scope of variables declared earlier in the file. Therefore, any variables declared before the **declare ingres** statement will not be visible to the preprocessor. For this reason, it is an error to issue two **declare ingres** statements in a single program module.

You should not issue the **declare ingres** statement in subroutines and functions declared to EQUEL. After processing a **sub** or **function** statement, the preprocessor automatically generates the **declare ingres** statement and terminates the scope of previous subprograms. If you do issue the **declare ingres** statement in a subroutine or function known to EQUEL, the preprocessor will generate a warning and ignore the statement. On the other hand, if you do not define a subprogram to EQUEL (perhaps because it lists formal parameters of a type unavailable to EQUEL variables), you must specifically issue the **declare ingres** statement before any EQUEL declarations or statements in that subprogram.

Because a **def** function is local to the program or subprogram that defines it, the **declare ingres** statement is neither needed nor automatically generated for it. The **def** function inherits its program module's variables and definitions.

Reserved Words in Declarations

All EQUEL keywords are reserved: therefore, you cannot declare variables with the same names as EQUEL keywords. You can only use them in quoted string literals. These words are:

byte	**case**	**com**	**common**	**constant**
decimal	**declare**	**def**	**dim**	**dimension**
double	**dynamic**	**external**	**fnend**	**function**
functionend	**group**	**integer**	**long**	**map**
real	**record**	**single**	**string**	**sub**
subend	**variant**	**word**		

The EQUEL preprocessor does not distinguish between uppercase and lowercase in keywords. In generating BASIC code, it converts any uppercase letters in keywords to lowercase.

Data Types

EQUEL/BASIC accepts elementary BASIC data types in variable declarations and maps them to their corresponding Ingres types as shown in the following table.

BASIC Data Types and Corresponding Ingres Type

BASIC Type	Ingres Type
string	**character**
integer	**integer**
long	**integer**
word	**integer**
byte	**integer**
real	**float**
single	**float**
double	**float**

EQUEL accepts the BASIC **record** type in variable declarations, providing the record has already been declared to EQUEL.

The following data types are illegal and will cause declaration errors:

gfloat
hfloat

Neither the preprocessor nor the runtime support routines support **gfloat** or **hfloat** floating-point arithmetic. You should not compile the BASIC source code with the command line qualifiers **gfloat** or **hfloat** if you intend to pass those floating-point values to or from Ingres objects.

The String Data Type

EQUEL accepts both fixed-length and dynamic **string** declarations. Strings can be declared to EQUEL using any of the declarations listed later. Note that you can indicate string length only for non-dynamic strings; that is, for string declarations appearing in **common**, **map** or **record** declarations. For example,

```
##   common (globals) string ename = 30
```

is acceptable, but

```
##   declare string bad_str_var = 30 ! length is illegal
```

will generate an error.

The reference to an uninitialized BASIC dynamic string variable in an embedded statement that assigns the value of that string to Ingres results in a runtime error because an uninitialized dynamic string points at a zero address. This restriction does not apply to the retrieval of data into an uninitialized dynamic string variable.`

The Integer Data Type

All BASIC **integer** data type sizes are accepted by the preprocessor. It is important that the preprocessor knows about integer size, because it generates code to load data in and out of program variables. EQUEL assumes that integer size is four bytes by default. However, you may inform EQUEL of a non-default integer size by using the **-i** flag on the preprocessor command line. (For more information, see Precompiling, Compiling and Linking an EQUEL Program in this chapter.)

For example, the preprocessor command:

```
$ eqb -i2 myfile.qb
```

causes the preprocessor to treat all variables of type **integer** as two-byte quantities. If you use the **-i** flag, be sure to inform the BASIC compiler of the integer size, either by means of an option to the **basic** command or, in the program, by means of the BASIC **options** statement.

You can explicitly override the default or the preprocessor **-i** size by using the BASIC subtype words **byte**, **word** or **long** in the variable declaration, as these examples illustrate:

```
##   declare byte one_byte_int
##   common (globals) word two_byte_int
##   external long four_byte_int
```

These declarations create EQUEL **integer** variables of one, two, and four bytes, respectively, regardless of the default setting.

An integer variable can be used with any numeric-valued object to assign or receive numeric data. For example, such a variable can be used to set a field in a form or to retrieve a column from a database table.

The Real Data Type

As with the **integer** data type, EQUEL must know the size of **real** data to manipulate variables of type **real**. Two sizes of **real** data are acceptable to EQUEL: four-byte variables (the default) and eight-byte variables. Again, you can change the default size with a flag on the preprocessor command line—in this case, the **-r** flag. For example:

```
$ eqb -r8 myfile.qb
```

instructs EQUEL to treat all **real** variables as eight-byte quantities. You can explicitly override the default or the **-r** size by using the BASIC subtype words **single** or **double** in a variable declaration. For example, the following two declarations

```
##   declare single four_byte_real
##   map (myarea) double eight_byte_real
```

create EQUEL **real** variables of four and eight bytes, respectively, regardless of the default setting.

A **real** variable can be used in EQUEL statements to assign or receive numeric data (both real and integer) to and from database columns, form fields and table field columns. It cannot be used to specify numeric objects, such as table field row numbers.

The Record Data Type

You can declare EQUEL variables with type **record** if you have already defined the record to EQUEL. Later sections discuss the syntax of EQUEL record definitions. You can also declare formal parameters of type **record** to subprograms. In that case, the EQUEL record definition must *follow* the EQUEL subprogram statement. Later sections discuss **record** type formal parameters.

Variable and Constant Declaration Syntax

EQUEL/BASIC variables and constants can be declared in a variety of ways when those statements are made known to EQUEL with the ## mark. The following sections describe these declaration statements and their syntax.

The Declare Statement

The **declare** statement for an EQUEL/BASIC variable has the following syntax:

declare *type identifier* [**(***dimensions***)**] {**,** [*type*] *identifier* [**(***dimensions***)**]}

The **declare** statement for an EQUEL/BASIC constant has the syntax:

declare *type* ***constant*** *identifier* **=** *literal* {**,** *identifier* = *literal*}

Syntax Notes:

1. If the word **constant** is specified, the declared constants cannot be targets of Ingres retrievals.
2. The *type* must be a BASIC type acceptable to EQUEL or, in the case of variables only, a **record** type already defined to EQUEL. Note that the type is mandatory for EQUEL declarations, because EQUEL has no notion of a default type. The type need only be specified once when declaring a list of variables of the same type.
3. The *dimensions* of an array specification are not parsed by the EQUEL preprocessor. Consequently, the preprocessor does not check bounds. Note also that an illegal dimension, such as a non-numeric value, will be accepted by the preprocessor, but will later cause BASIC compiler errors.
4. You cannot use the **declare** statement to declare **def** functions to EQUEL.

The following example illustrates the use of the **declare** statement:

```
##   declare integer enum, eage, string ename
##   declare single constant minsal = 12496.62
##   declare real esal(100)
##   declare word null_ind       ! Null indicator
```

The Dimension Statement

The **dimension** statement can be used to declare arrays to EQUEL. Its syntax is:

dimension | **dim** *type identifier****(****dimensions****)*** {**,** [*type*] *identifier* ***(****dimensions****)***}

Syntax Notes:

1. The *type* must be a BASIC type acceptable to EQUEL or a record already defined to EQUEL. Note that the type is mandatory for EQUEL declarations, because EQUEL has no notion of a default type. The type need only be specified once when declaring a list of variables of the same type.
2. The *dimensions* of an array specification are not parsed by the EQUEL preprocessor. Consequently, the preprocessor does not check bounds. Note also that an illegal dimension, such as a non-numeric value, will be accepted by the preprocessor, but will later cause BASIC compiler errors. Furthermore, EQUEL does not distinguish between executable and declarative **dimension** statements. If you have used the **dimension** statement to declare an executable array to EQUEL, using the EQUEL ## mark with subsequent executable dimension statements of the same array will cause a redeclaration error.

The following example illustrates the use of the **dimension** statement:

```
##  dim string employee_names(100,20)  ! declarative DIM statement
##  dimension long emp_id(100,2,2)
##  dimension double expenses(numdepts) ! executable DIM statement
```

Static Storage Variable Declarations

EQUEL supports the BASIC **common** and **map** variable declarations. The syntax for a **common** variable declaration is as follows:

common | **com** [**(***com_name***)**]
type identifier [**(***dimensions***)**] [**=** *str_length*]
{**,** [*type*] *identifier* [**(***dimensions***)**] [**=** *str_length*]}

The syntax for a **map** variable declaration is as follows:

map | **map dynamic (***map_name***)**
type identifier [**(***dimensions***)**] [**=** *str_length*]
{**,** [*type*] *identifier* [**(***dimensions***)**] [**=** *str_length*]}

Syntax Notes:

1. The *type* must be a BASIC type acceptable to EQUEL or a **record** type already defined to EQUEL. Note that the type is mandatory for EQUEL declarations, because EQUEL has no notion of a default type. The type need only be specified once when declaring a list of variables of the same type.
2. The *dimensions* of an array specification are not parsed by the EQUEL preprocessor. Consequently, the preprocessor does not check bounds. Note also that an illegal dimension, such as a non-numeric value, will be accepted by the preprocessor, but will later cause BASIC compiler errors.

3. The string length, if present, must be a simple integer literal.
4. The *com_name* or *map_name* clause is not parsed by the EQUEL preprocessor. Consequently, the preprocessor will accept common and map areas of the same name in a single program module. It will also accept a **map dynamic** statement whose *com_name* has not appeared in another **map** statement. Either of these situations will later cause BASIC compiler errors.

The following example uses **common** and **map** variable declarations:

```
##   common (globals) string address = 30, integer zip
##   map (ebuf) byte eage, string ename = 20,    single
##          emp_num
##   common (globals) integer empid (200)
```

The External Statement

You can inform EQUEL of variables and constants declared in an external module. The syntax for a variable is as follows:

external *type identifier* {**,** *identifier*}

The syntax for a constant is as follows:

external *type* **constant** *identifier* {**,** *identifier*}

Syntax Notes:

1. EQUEL applies the same restrictions on *type* as VMS BASIC.
2. You cannot declare external functions or subroutines to EQUEL. EQUEL understands only function and subroutine *definitions*.

The following example illustrates the use of the **external** statement:

```
##   external integer empform, infoform  !Compiled forms
##   external single constant emp_minsal
```

Parameter Variables

Variables can be declared by listing them as formal parameters to a subroutine or function definition, providing the **sub**, **function**, or **def** statement is preceded by the EQUEL ## mark. The syntax for a **function** statement is:

function *type identifier* [*pass_mech*]
[**(***type identifier* [**(***dimensions***)**] [**=** *str_length*] [*pass_mech*]
{**,** **[***type*] *identifier* [**(***dimensions***)**] [**=** *str_length*]
[*pass_mech*]}**)**]

The **sub** statement has the syntax:

sub *identifier* [*pass_mech*]
[**(***type identifier* [**(***dimensions***)**] [**=** *str_length*] [*pass_mech*]
{**,** [*type*] *identifier* [**(***dimensions***)**] [**=** *str_length*]
[*pass_mech*]}**)**]

The **def** statement has the syntax:

def *type identifier*
[**(***type identifier* {**,** [*type*] *identifier*}**)**]

Syntax Notes:

1. The *type* must be a BASIC type acceptable to EQUEL or a BASIC **record**. Unlike the rules for other EQUEL variable declarations, you can define the record to EQUEL *after* it appears in the parameter list. For example:

```
##   sub process_info (emp_rec emp)
##      record emp_rec
##          ...
##      end record emp_rec
```

2. The type is mandatory for EQUEL parameter declarations because EQUEL has no notion of a default type. The type need only be specified once when declaring a list of parameters of the same type.
3. The *pass_mech* (allowed on **sub** and **function** statements) may be **by desc** or **by ref**. However, the preprocessor does not verify that the formal parameter declaration is consistent with the passing mechanism. You should follow the VMS BASIC rules for parameter passing mechanisms.
4. The *dimensions* of an array specification are not parsed by the EQUEL preprocessor. Consequently, the preprocessor does not check bounds. Note also that an illegal dimension, such as a non-numeric value, will be accepted by the preprocessor, but will later cause BASIC compiler errors.

The following example illustrates the use of parameter variables:

```
##   def real newsal (integer grade, real oldsal, single percent)
##   function string get_addr by ref (string ename = 20, integer eno)
##   sub new_emps (integer deptno, string emplist (100) = 20 by ref)
```

Record Type Definitions

EQUEL accepts BASIC record definitions. The syntax of a record definition is:

record *identifier*
record_component
{*record_component*}
end record [*identifier***]**

where *record_component* can be any of the following:

type identifier [**(***dimensions***)**] [**=** *str_length*]
{**,** [*type*] *identifier* [**(***dimensions***)**] [**=** *str_length*]}

group_clause

variant_clause

host_code

In turn, the syntax of a *group_clause* is:

group *identifier* [**(***dimensions***)**]
record_component
{*record_component*}
end group [*identifier*]

The syntax of a *variant_clause* is:

variant
case_clause
{*case_clause*}
end variant

where *case_clause* consists of:

case
record_component

Syntax Notes:

1. The *type* must be a BASIC type acceptable to EQUEL or a **record** type already defined to EQUEL. Note that the type is mandatory for EQUEL declarations because EQUEL has no notion of a default type. The type need only be specified once when declaring a list of variables of the same type.
2. The string length clause is allowed only for record components of type **string**.
3. The host code record component allows you to declare components of records without informing EQUEL of their existence. For instance, you may want to declare **fill** items or components whose type is not allowed in an EQUEL declaration. For example, the following record definition is acceptable to EQUEL:

```
##   record dept_rec
##      double net_profit
         gfloat gross_sales  ! Not for use with EQUEL statements
##   end record
```

4. Record definitions must appear before declarations using that record type. An exception occurs where a parameter to an EQUEL subroutine or function is of the **record** type. In that case, you may define the record to EQUEL after declaring it in the parameter list.

The following example illustrates the use of record type definitions:

```
##  record emp_history
##          string ename = 30
##          group prev_employers(10)
##              string comp_name = 30
##              real salary
##              integer num_years
##  end group prev_employers
##  end record emp_history

##  record emp_sports
##          string ename = 30
##          variant
##              case
##                      group golf
##                              integer handicap
##                              string club_name
##                      end group golf
##              case
##                      group baseball
##                              integer batting_avg
##                              string team_name
##                      end group baseball
##              case
##                      group tennis
##                              integer seed
##                              string club_name
##                      end group tennis
##          end variant
##  end record emp_hobbies
```

The Indicator Variable

An *indicator variable* is a 2-byte integer variable. There are three possible ways to use these in an application:

- In a statement that retrieves data from Ingres, you can use an indicator variable to determine if its associated host variable was assigned a null.
- In a statement that sets data to Ingres, you can use an indicator variable to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator variable as a check that the associated host variable is large enough to hold the full length of the returned character string.

An indicator variable can be declared using the integer **word** subtype, or, if the **-i2** preprocessor command line flag was used, can be declared as an **integer**. The following example declares two indicator variables, one a single variable and the other an array of indicators:

```
##   declare word ind, ind_arr(10)
```

Assembling and Declaring External Compiled Forms

You can pre-compile your forms in the Visual Forms Editor (VIFRED). This saves time that would otherwise be required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO. After the file is created, you can use the following VMS command to assemble it into a linkable object module:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the Embedded SQL/FORMS statement **addform** can refer to this global object, you must declare it in an Embedded SQL declaration section, using the following syntax:

external integer *formname*

Syntax Notes:

1. The *formname* is the actual name of the form. VIFRED gives this name to the address of the global object. The *formname* is also used as the title of the form in other EQUEL/FORMS statements. In all statements that use the *formname* as an argument, except for **addform**, you must dereference the name with the **#** sign.
2. The **external** statement associates the object with the external form definition.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
##   external integer empform
##   addform empform            ! The global object
##   display #empform           ! The name of the form
```

Concluding Example

The following example demonstrates some simple EQUEL/BASIC declarations:

```
##   declare ingres
##   declare byte        d_byte ! Variables of each data type
##   declare word        d_integer2
##   declare long        d_integer4
##   declare integer     d_integer_def
##   declare single      d_real4
##   declare double      d_real8
##   declare real        d_real_def
##   declare decimal(6,2) d_decimal
##   declare string d_string
##   declare integer constant num_depts = 10 ! Constant
```

```
##   common(globs) real e_raise ! Static storage variables
##   map (ebuf) string ename = 20
##   dim string emp_names(100,30) ! Array declarations
##   declare integer dept_id(10)
##   common(globs) string e_address(40) = 30

##   record person ! Variant record
##       byte age
##       long flags
##       variant
##          case
##              group emp_list
##                       string full_name = 30
##              end group
##          case
##              group emp_directory
##                       string firstname = 12
##                       string lastname = 8
##              end group
##      end variant
##   end record

##   declare person p_table(100)         ! Array of records
##   external integer empform, deptform  ! Compiled forms
##   dim word indicators(10)             ! Array of null indicators
```

The Scope of Variables

Variable names must be unique in their scope. Variable names are local to the program module in which they are declared.

The scope of a variable opens at its declaration. Generally, its scope remains open until the end of the program module. For example, an EQUEL variable declared in a main program will be visible to all subsequent EQUEL statements until a BASIC **end**, **sub**, or **function** statement is processed by EQUEL. (Remember that the preprocessor will process these statements only if they are preceded by the EQUEL ## mark.) Similarly, an EQUEL variable declared in a **sub** or **function** subprogram or a formal parameter to that subprogram will be visible until the **end sub** or **end function** statement is processed by EQUEL. Processing of another **sub** or **function** statement would also close the scope of the subprogram.

Note that scoping rules for **def** functions differ somewhat for EQUEL and BASIC. The scope of the formal parameters to an EQUEL **def** function remains open until the **end def** statement is processed. The same is true of variables declared in the **def** function. In other words, EQUEL treats such variables as local variables. However, while BASIC also regards *parameters* as local to the **def** function, it allows *variables* declared in the **def** function to have a global scope. If you wish an EQUEL variable to have a scope that is global to the program module as a whole, you must declare it in the program module, *not* in a **def** function definition.

In order to ensure that EQUEL follows the same scoping conventions as those followed by BASIC, you should observe these rules:

- Always use the EQUEL ## mark on **sub** and **function** statements, even if the parameters are not EQUEL variables. These statements cause EQUEL to open a new scope, closing off all previous scopes.
- Be aware that the **## declare ingres** statement closes off previously opened scopes and opens a new scope. Therefore, if you do issue this statement, you should include it before any EQUEL declarations in your main program.
- If you declare a **def** function statement to EQUEL, you must also issue the **end def** statement to EQUEL, so that it may close the local scope of **def** variables and parameters.
- Issue the **end**, **end sub** and **end function** statements to EQUEL in order that the preprocessor may be fully aware of the scoping of EQUEL variables. These statements must each appear on a line by themselves, with no comments separating the keywords.

The following example illustrates the scope of variables in an EQUEL/BASIC program.

```
10  ##  declare ingres
    ##      common (glob) integer a, real b
            declare single c, double d
            declare double function xyz(single)
            ! Visible to EQUEL: a, b
            ! Visible to BASIC: a, b, c, d, xyz
            ...
    ##  def double xyz(single e)
    ##      declare byte f
            declare string g
            ! Visible to EQUEL: a, b, e, f
            ! Visible to BASIC: a, b, c, d, e, f, g, xyz
            ...
    ##  end def
        ! Visible to EQUEL: a, b
        ! Visible to BASIC: a, b, c, d, f, g, xyz
        d = xyz(c)
        ...
        call uvw(d)
    ##  end
        ! Visible to EQUEL: no variables
        ! Visible to BASIC: no variables
20  ##  sub uvw(double p)
        ! No DECLARE Ingres statement needed in subprogram
    ##          common (glob) integer a, real b
    ##          declare byte q
                ! Visible to EQUEL: p, a, b, q
                ! Visible to BASIC: p, a, b, q
                ...
    ##  end sub
        ! Visible to EQUEL: no variables
        ! Visible to BASIC: no variables
```

Special care should be taken when using variables in a **declare cursor** statement. The variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued, and, at that time, evaluates any associated variables. For example, in the following program fragment, even though the variable "number" is valid to the preprocessor at the point of the **declare cursor** statement, it is not an explicitly declared variable name for the BASIC compiler at the point that the **open** is issued, possibly resulting in a runtime error. Because BASIC allows implicit variable declarations (although EQUEL does not), the compiler itself will not, however, generate an error message.

```
1   ##  sub Init_Csr            ! Example contains an error
    ##      declare integer number
            ! Cursor declaration includes reference to "number"
    ##      declare cursor c1 for
    ##          retrieve (employee.name, employee.age)
    ##          where employee.num = number
            ...
    ##  end sub

2   ##  sub Process_Csr
    ##      declare string ename
    ##      declare integer eage
            ! Opening the cursor evaluates invalid "number"
    ##      open cursor c1
    ##      retrieve cursor c1 (ename, eage)
            ...
    ##  end sub
```

Variable Usage

BASIC variables declared to EQUEL can substitute for most elements of EQUEL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. To use a BASIC variable in an EQUEL statement, just use its name. To refer to an element, such as a database column, with the same name as a variable, dereference the element with the EQUEL dereferencing indicator (**#**). As an example of variable usage, the following **retrieve** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idnovar" as an expression in the **where** clause:

```
##   retrieve (namevar = e.name, numvar = e.num)
##          where e.idno = idnovar;
```

You must verify that the statement using the variable is in the scope of the variable's declaration. Various rules and restrictions apply to the use of BASIC variables in EQUEL statements. The sections below describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

A simple scalar-valued variable (integer, floating-point or character string) is referred to by the syntax:

simplename

Syntax Notes:

1. If the variable is used to send data to Ingres, it can be a scalar-valued variable or constant.
2. If the variable is used to receive data from Ingres, it cannot be a variable declared as a constant.
3. The reference to an uninitialized BASIC dynamic string variable in an embedded statement that assigns the value of that string to Ingres results in a runtime error because an uninitialized dynamic string points at a zero address. This restriction does not apply to the retrieval of data into an uninitialized dynamic string variable.

The following program fragment demonstrates a typical message-handling routine that uses two scalar-valued variables, "buffer" and "seconds":

```
##   sub Msg (string buffer, integer seconds)
##      message buffer
##      sleep seconds
##   end sub
```

Array Variables

An array variable is referred to by the syntax:

arrayname*(subscript{,subscript})*

Syntax Notes:

1. The variable must be subscripted, because only scalar-valued elements (integers, floating-point and character strings) are legal EQUEL values.
2. When the array is declared, the array bounds specification is not parsed by the EQUEL preprocessor. Consequently, illegal bounds values will be accepted. Also, when an array is referenced, the subscript is not parsed, allowing illegal subscripts to be used. The preprocessor only confirms that an array subscript is used for an array variable. You must make sure that the subscript is legal and that the correct number of indices are used.

Record Components

The syntax EQUEL uses to refer to a record component is the same as in BASIC:

record**::***component{::component}*

Syntax Notes:

1. The last record *component* denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and records, but the last object referenced must be a scalar value. Thus, all the following references are legal:

```
! Assume correct declarations for "employee", "person"
! and other records.
    employee::sal                ! Component of a record
    person(3)::pname             ! Component of an element of an array
    rec1::mem1::mem2::age        ! Deeply nested component
```

2. All record components must be fully qualified when referenced. Elliptical references, such as references that omit group names, are not allowed.

The example below uses the array of records "emprec" to load values into the tablefield "emptable" in form "empform."

```
##  record Employee_Rec
##              string  ename = 20
##              word    eage
##              integer eidno
##              string  ehired = 25
##              string  edept = 10
##              real    esalary
##  end record

##  declare Employee_Rec emprec(100)
    declare integer i

    . . .

    for i = 1 to 100
##          loadtable empform emptable
##      (ename = emprec(i)::ename,
##       eage = emprec(i)::eage, eidno = emprec(i)::eidno,
##       ehired = emprec(i)::ehired,
##       edept = emprec(i)::edept,
##       esalary = emprec(i)::esalary)
    next i
```

Using Indicator Variables

The syntax for referring to an *indicator* variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

host_variable:indicator_variable

Syntax Note:

The indicator variable can be a simple variable, an array element or a record member that yields a 2-byte integer (the **word** subtype). For example:

```
declare word ind_var, ind_arr(5)
var_1:ind_var
var_2:ind_arr(2)
```

Data Type Conversion

A BASIC variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into string variables.

Data type conversion occurs automatically for different numeric types such as from floating-point Ingres database column values into integer BASIC variables, and for character strings, such as from varying-length Ingres character fields into fixed-length BASIC string buffers.

Ingres does *not* automatically convert between numeric and character types. You must use the Ingres type conversion operators, the Ingres **ascii** function, or a BASIC conversion procedure for this purpose.

The following table shows the default type compatibility for each Ingres data type. Note that some BASIC types do not match exactly and, consequently, may go through some runtime conversion.

Ingres Data Types and Corresponding BASIC Types

Ingres Type	BASIC Type
c(*N***)**, **char**(*N*)	**string** (dynamic)
c(*N*), **char(*N*)**	**string** (static with length clause of *N*)
text(*N*), **varchar(*N*)**	**string** (dynamic)
text(*N*), **varchar**(*N*)	**string** (static with length clause of *N*)
i1, integer1	**integer byte**
i2, integer2	**integer word**
i4, integer4	**integer long**
f4, float4	**real single**
f8, float8	**real double**
date	**string** (dynamic)

Ingres Type	BASIC Type
date	**string** (static with length clause of 25)
money	**real double**

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and forms system and numeric BASIC variables. The standard type conversion rules (according to standard VAX rules) are followed. For example, if you assign a **real** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion.

The Ingres **money** type is represented as **real double**, an 8-byte floating-point value.

Runtime Character Type Conversion

Automatic conversion occurs between Ingres character string values, database columns of type **c**, **char**, **text** or **varchar**, and form fields of type **character**. Several considerations apply when dealing with string conversions, both to and from Ingres.

The conversion of BASIC string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables because the blanks make no sense in that context. For example, the string literals "empform " and "empform" refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **c** or **character**, a database column of type **text** or **varchar**, or a **character** form field. Ingres pads columns of type **c** or **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **text** or **varchar**, or in form fields.

Second, the BASIC convention is to blank-pad static character strings. For example, the character string "abc" may be stored in a BASIC static string variable of length 5 as the string "abc " followed by two blanks.

When retrieving character data from an Ingres database column or form field into a BASIC variable, take note of the following conventions:

- When character data is retrieved from Ingres into a BASIC static string variable and the variable is longer than the value being retrieved, the variable is padded with blanks. If the variable is shorter than the value being retrieved, the value is truncated. You should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data.
- When character data is retrieved into a BASIC dynamic string variable, the variable's new length will exactly match the length of the data retrieved. Ingres manipulates dynamic strings in exactly the same way as BASIC does, creating and modifying storage requirements as necessary. For example, when zero-length **varchar** data is retrieved into a BASIC dynamic string variable, storage will not be created for the string.

When inserting character data into an Ingres database column or form field from a BASIC variable, note the following conventions:

- When data is inserted from a BASIC variable into a database column of type **c** or **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a BASIC variable into a database column of type **text** or **varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **text** or **varchar** column. For example, when a string "abc" stored in a BASIC static string variable of length 5 as "abc " is inserted into the **text** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the EQUEL **notrim** function. It has the following syntax with *stringvar* as a character string variable.

  notrim(*stringvar)*

  When used with **repeat** queries, the **notrim** syntax is:

  @**notrim(***stringvar***)**

  If the **text** or **varchar** column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a BASIC variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

You cannot use zero-length or uninitialized BASIC dynamic strings in **insert** or **update** statements. This is because an uninitialized dynamic string has no storage allocated for it and Ingres treats it as a non-existent variable.

When comparing character data in an Ingres database column with character data in a BASIC variable, note the following convention:

- When comparing data in **c**, **character**, or **varchar** database columns with data in a character variable all trailing blanks are ignored. Trailing blanks are significant in **text**. Initial and embedded blanks are significant in **character**, **text**, and **varchar**; they are ignored in **c**.

Note: The conversion of character string data between Ingres objects and BASIC variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. The *QUEL Reference Guide* has information on blanks when comparing with the various Ingres character types.

The Ingres **date** data type is represented as a 25-byte string.

The example below uses the **notrim** function and the truncation rules explained above.

```
11  ##  sub Notrim_Test
        !
        ! Assume a table called "textchar" has been
        ! created with the following CREATE statement:
        !
        ! CREATE textchar
        !        (row = integer,
        !        data = text(10))        Note the text type
        !

    ##  declare word    row
    ##  common string   sdata = 7       ! Static string
    ##  declare string  ddata           ! Dynamic string

        sdata = 'abc  '                 ! Holds "abc " with 4 blanks
        ddata = 'abc'                   ! Holds "abc"

        ! This APPEND adds string "abc" (blanks truncated)
    ##  append to textchar (#row = 1, data = sdata)

        ! This APPEND adds string "abc" (never had blanks)
    ##  append to textchar (#row = 2, data = ddata)

        ! This APPEND adds string "abc ", with tailing
        ! blanks left intact by using the NOTRIM function.
    ##  append to textchar
    ##      (#row = 3, data = NOTRIM(sdata))
```

```
    ! This RETRIEVE retrieves rows #1 and #2, because
    ! trailing blanks were suppressed when those rows
    ! were inserted.
##  range of t IS textchar
##  retrieve (row = t.#row) WHERE LENGTH(t.data) = 3
##  {
        print 'Row found =', row
##  }

    print '---------'

    ! This RETRIEVE retrieves row #3, because the
    ! NOTRIM function left trailing blanks in the
    ! "sdata" variable
    ! in the last APPEND statement.
##  retrieve (row = t.#row) WHERE LENGTH(t.data) = 7
##  {
        print 'Row found =', row
##  }

##  end sub
```

Dynamically Built Param Statements

The **param** feature dynamically builds EQUEL statements. EQUEL/BASIC does not currently support **param** versions of statements. **Param** statements are supported in EQUEL/C and EQUEL/Fortran.

Runtime Error Processing

This section describes a user-defined EQUEL error handler.

Programming for Error Message Output

By default, all Ingres and forms system errors are returned to the EQUEL program, and default error messages are printed on the standard output device. As discussed in the *QUEL Reference Guide* and the *Forms-based Application Development Tools User Guide*, you can also detect the occurrences of errors by means of the program using the **inquire_ingres** and **inquire_frs** statements. (Use the latter for checking errors after forms statements—see the examples in the *Forms-based Application Development Tools User Guide.* Use **inquire_ingres** for all other EQUEL statements.)

This section discusses an additional technique that enables your program not only to detect the occurrences of errors, but also to suppress the printing of default Ingres error messages if you choose. The **inquire** statements detect errors but do not suppress the default messages.

This alternate technique entails creating an error-handling function in your program and passing its address to the Ingres runtime routines. Then Ingres will automatically invoke your error handler whenever an Ingres or a forms-system error occurs. Your program error handler must be declared as follows:

```
function integer funcname (errno)
...
end function
```

This function must be passed to the EQUEL routine **IIseterr( )** for runtime bookkeeping using the statement:

```
call IIseterr(funcname)
```

This forces all runtime Ingres errors through your function, passing the error number as an argument. If you choose to handle the error locally and suppress Ingres error message printing, the function should return 0; otherwise the function should return the Ingres error number received. The error-handling function must return a **long** integer. If your default integer size is less than 4 bytes, you must declare the function to be a **long** function.

Avoid issuing any EQUEL statements in a user-written error handler defined to **IIseterr**, except for informative messages, such as **message, prompt, sleep** and **clear screen,** and messages that close down an application, such as **endforms** and **exit.**

The example below demonstrates a typical use of an error function to warn users of access to protected tables. This example passes through all other errors for default treatment.

```
1   ##  declare ingres
        external integer Err_Trap

    ##  Ingres personnel
        call IIseterr(Err_Trap)

        ...
    ##  exit
        end

2       function integer Err_Trap(integer ingerr)

            ! Error number for protected tables
            declare integer constant TBLPROT = 5003%

         if (ingerr = TBLPROT) then
            print 'You are not authorized for this operation'
            Err_Trap = 0%               ! Do not print messages
          else
                Err_Trap = ingerr       ! Ingres will print error
         end if
      end function
```

Precompiling, Compiling and Linking an EQUEL Program

This section describes the EQUEL preprocessor for BASIC and the steps required to precompile, compile, and link an EQUEL program.

Generating an Executable Program

Once you have written your EQUEL program, it must be preprocessed to convert the EQUEL statements into BASIC code. These sections describe the use of the EQUEL preprocessor. Additionally, it describes how to compile and link the resulting code to obtain an executable file.

The EQUEL Preprocessor Command

The BASIC preprocessor is invoked by the following command line:

eqb {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by EQUEL. The source file name, line number, and the erroneous statement itself are printed along with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If the **-f** flag is specified without a *filename*, the output is sent to standard output, one screen at a time. If the **-f** flag is omitted, output is given the basename of the input file, suffixed ".bas".
-i*N*	Sets the default size of ***integers*** to *N* bytes. *N* must be 1, 2, or 4. The default setting is 4.
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named filename***.lis***, where *filename* is the name of the input file.
-lo	Similar to **-l**, but the generated BASIC code also appears in the listing file.
-n. *ext*	Specifies the extension used for filenames in **## include** and ## include inline statements in the source code. If **-n** is omitted, include filenames in the source code must be given the extension ".qb".

Flag	Description
-o	Directs the preprocessor not to generate output files for include files. This does not affect the translated **include** statements in the main program. The preprocessor will generate a default extension for the translated include file statements unless you use the **-o**.ext flag.
-o.*ext*	Specifies the extension given by the preprocessor to both the translated **include** statements in the main program and the generated output files. If this flag is not provided, the default extension is ".bas". If you use this flag in combination with the **-o** flag, then the preprocessor generates the specified extension for the translated include statements, but does not generate new output files for the include statements.
-r*N*	Sets default size of **reals** to *N* bytes. *N* must be 4 or 8. The default setting is 4.
-s	Reads input from standard input and generates BASIC code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called "stdin.lis." To terminate the interactive session, type **Ctrl Z**.
-w	Prints warning messages.
-?	Shows what command line options are available for **eqb**.

The EQUEL/BASIC preprocessor assumes that input files are named with the extension ".qb". This default can be overridden by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated BASIC statements with the same name and the extension ".bas".

If you enter the command without specifying any flags or a filename, INGRES displays a flags list for the command.

The following table presents the options available with **eqb**.

Eqb Command Examples

Command	Comment
eqb file1	Preprocesses "file1.qb" to "file1.bas"
eqb -l file2.xb	Preprocesses "file2.xb" to "file2.bas" and creates listing "file2.lis"
eqb -s	Accepts input from standard input and writes generated code to standard output
eqb -ffile3.out file3	Preprocesses "file3.qb" to "file3.out"
eqb	Displays a list of flags available for this command.

The BASIC Compiler

As mentioned above, the preprocessor generates BASIC code. You should use the VMS **basic** command to compile this code. Most of the basic command line options can be used. You should not use the **g_float** or **h_float** qualifiers if floating-point values in the program are interacting with INGRES floating-point objects. If you use the **byte** or **word** compiler qualifiers, you must run the EQUEL preprocessor with the **-i1** or **-i2** flag. Similarly, use of the BASIC **double** qualifier requires that you have preprocessed your EQUEL file using the **-r8** flag. Note, too, that many of the statements that the EQUEL preprocessor generates are BASIC language extensions provided by VAX/VMS. Consequently, you should not attempt to compile with the **ansi_standard** qualifier.

The following example preprocesses and compiles the file "test1." Note that both the EQUEL preprocessor and the BASIC compiler assume the default extensions.

```
$ eqb test1
$ basic/list test1
```

Note: Check the Readme file for any operating system specific information on compiling and linking EQUEL/BASIC programs.

Linking an EQUEL Program

EQUEL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an EQUEL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
$ link dbentry.obj,-
  ii_system:[ingres.files]equel.opt/opt
```

It is recommended that you do not explicitly link in the libraries referenced in the EQUEL.OPT file. The members of these libraries change with different releases of INGRES. Consequently, you may be required to change your link command files in order to link your EQUEL programs.

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *QUEL Reference Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry") by listing it in the link command, as in the following example:

```
$ link formentry,-
  empform.obj,-
  ii_system:[ingres.files]equel.opt/opt
```

Linking an EQUEL Program without Shared Libraries

While the use of shared libraries in linking EQUEL programs is recommended for optimal performance and ease-of-maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by EQUEL are listed in the esql.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
$ link dbentry,-
  ii_system:[ingres.files]equel.noshare/opt
```

Include File Processing

The EQUEL **include** statement provides a means to include external files in your program's source code. Its syntax is:

include *filename*

Filename is a quoted string constant specifying a file name, or a logical name that points to the file name. You must use the default extension ".qb" on names of **include** files unless you override this requirement by specifying a different extension with the **-n** flag of the **eqb** command.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *QUEL Reference Guide*.

The included file is preprocessed and an output file with the same name but with the default output extension ".bas" is generated. You can override this default output extension with the **-o*.ext*** flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified include output file. If the **-o** flag is used (with no extension), no output file is generated for the include file. This is useful for program libraries that are using VMS MMS dependencies.

If you use both the **-o.***ext* and the **-o** flags, then the preprocessor will generate the specified extension for the translated **include** statements in the program but will not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## include 'employee.qb'
```

is preprocessed to the BASIC statement:

```
% include "employee.bas"
```

and the file "employee.qb" is translated into the BASIC file "employee.bas."

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
## include 'mydecls'
```

The name "mydecls" is defined as a system logical name pointing to the file "dra1:[headers]myvars.qb" by means of the following command at the DCL level:

```
$ define mydecls dra1:[headers]myvars.qb
```

Assume now that "inputfile" is preprocessed with the command:

```
$ eqb -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the BASIC statement:

```
% include "dra1:[headers]myvars.h"
```

and the BASIC file "dra1:[headers]myvars.h" is generated as output for the original include file, "dra1:[headers]myvars.qb." See the *QUEL Reference Guide* for including source code using the **include inline** statement.

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.qb," the EQUEL statement:

```
## include '[-.headers]myvars.qb'
```

is preprocessed to the BASIC statement:

```
% include "[-.headers]myvars.bas"
```

and the BASIC file "dra1:[headers]myvars.bas," is generated as output for the original include file, "dra1:[headers]myvars.qb."

Including Source Code with Labels

Some EQUEL statements generate labels in the output code. If you include a file containing such statements, you must be careful to include the file only once in a given BASIC scope. Otherwise, you may find that the compiler later complains that the generated labels are defined more than once in that scope.

The statements that generate labels are the **retrieve** statement and all the EQUEL/FORMS block-type statements, such as **display** and **unloadtable**.

Coding Requirements for Writing EQUEL Programs

The following sections describe coding requirements for writing EQUEL programs.

Comments Embedded in BASIC Output

Each EQUEL statement generates one comment and a few lines of BASIC code. You may find that the preprocessor translates 50 lines of EQUEL into 200 lines of BASIC. This may result in confusion about line numbers when you are debugging the original source code. To facilitate debugging, each group of BASIC statements associated with a particular statement is preceded by a comment corresponding to the original EQUEL source. (Note that only *executable* EQUEL statements are preceded by a comment.) Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

Embedding Statements Inside BASIC If Blocks

The preprocessor never generates line numbers as its own. Therefore, you can enclose EQUEL statements in the **then** or **else** clauses of a BASIC **if** statement without changing program control.

For example:

```
    if (error = 1%) then
##      message "Error on update"
##      sleep 2
    endif
```

An EQUEL Statement that Does Not Generate Code

The **declare cursor** statement does not generate any BASIC code. This statement should not be coded as the only statement in BASIC constructs that does not allow *empty* statements.

EQUEL/BASIC Preprocessor Errors

To correct most errors, you may wish to run the EQUEL preprocessor with the listing (**-l**) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to the BASIC language, see the next section.

Preprocessor Error Messages

The following is a list of error messages specific to the Ada language.

E_E60001 "The ADA variable '%0c' is an array and must be subscripted."

Explanation: A variable declared as an array must be subscripted when referenced. The preprocessor does not confirm that you use the correct number of subscripts. A variable declared as a 1-dimensional array of characters, must not be subscripted as it refers to a character string.

E_E60002 "The ADA variable '%0c' is not an array and must not be subscripted."

Explanation: A variable not declared as an array cannot be subscripted. You cannot subscript string variables in order to refer to a single character or a slice of a string (substring).

E_E60003 "The ADA identifier '%0c' is not a declared type."

Explanation: The identifier was used as an Ada type name in an object or type declaration. This identifier has not yet been declared to the preprocessor and is not a preprocessor-predefined type name.

E_E60004 "The ADA CHARACTER variable '%0c' must be a 1-dimensional array."

Explanation: Variables of type CHARACTER can only be declared as 1-dimensional arrays. You cannot use a single character or a multidimensional array of characters as an Ingres string. Note that you can use a multidimensional array of type STRING.

E_E60005 "The ADA DIGITS clause '%0c' is out of the range 1..16."

Explanation: Embedded Ada supports D_FLOAT floating point variables. Consequently, all DIGITS specifications must be in the specified range.

E_E60006 "Statement '%0c' is embedded in INCLUDE file package specification."

Explanation: Preprocessor INCLUDE files may only be used for Ada package specifications. The preprocessor generates an Ada WITH clause for the package. No executable statements may be included in the file because the code generated will not be accepted by the Ada compiler in a package specification.

E_E60007 "Too many names (%0c) in ADA identifier list. Maximum is %1c."

Explanation: Ada identifier lists cannot have too many names in the comma-separated name list. The name specified in the error message caused the overflow, and the remainder of the list is ignored. Rewrite the declaration so that there are fewer names in the list.

E_E60008 "The ADA identifier list has come up short."

Explanation: The stack used to store comma separated names in Ada declarations has been corrupted. Try rearranging the list of names in the declaration.

E_E60009 "The ADA CONSTANT declaration of '%0c' must be initialized."

Explanation: CONSTANT declarations must include an initialization clause.

E_E6000A "The ADA identifier '%0c' is either a constant or an enumerated literal."

Explanation: The named identifier was used to retrieve data from Ingres. A constant, an enumerated literal and a formal parameter with the IN mode are all considered illegal for the purpose of retrieval.

E_E6000B "The ADA variable '%0c' with '.ALL' clause is illegal."

Explanation: The ADA .ALL clause, as specified with access objects, can be used only if the variable is an access object pointing at a single scalar-valued type. If the type is not scalar valued, or if the access object is pointing at a record or array, then the use of .ALL is illegal.

E_E6000C "The ADA variable '%0c' with '.ALL' clause is not a scalar type."

Explanation: The Ada .ALL clause, as specified with access objects, can be used only if the variable is an access object pointing at a single scalar-valued type. If the type is not scalar valued, or if the access object is pointing at a record or array, then the use of .ALL is illegal.

E_E6000D "Last component in ADA record qualification '%0c' is illegal."

Explanation: The last component referenced in a record qualification is not a member of the record. If this component was supposed to be declared as a record, the following components will cause preprocessor syntax errors.

E_E6000E "In ADA RENAMES statement, '%0c' must be a constant or a variable."

Explanation: The target object of a RENAMES statement must be a constant or a variable, and the item being declared is used a synonym for the target object.

E_E6000F "In ADA RENAMES statement, object is incompatible with type."

Explanation: The type of the target object in the RENAMES statement must be compatible in base type, size and array dimensions with the type name specified in the declaration.

E_E60010 "Only one name may be declared in an Ada RENAMES statement."

Explanation: One object can rename only one other object.

E_E60011 "Unclosed ADA block. There are %0c block(s) left open."

Explanation: If a file is terminated early or the END statement closing an Ada compilation unit is missing, this error will occur. If syntax errors were issued while parsing the compilation unit header, correct those errors first.

E_E60012 "The ADA variable '%0c' has not been declared."

Explanation: The named identifier was used where a variable must be used to set or retrieve Ingres data. The variable has not yet been declared.

E_E60013 "The ADA type %0c is not supported."

Explanation: Some Ada types are not supported because they are not compatible with the Ingres runtime system.

E_E60014 "The ADA variable '%0c' is a record, not a scalar value."

Explanation: The named variable qualification refers to a record. It was used where a variable must be used to set or retrieve Ingres data. This error may also cause syntax errors on record component references.

E_E60015 "You must issue a '## WITH %0c' before statement '%1c'."

Explanation: If your compilation unit includes forms statements you must issue the WITH EQUEL_FORMS clause. Otherwise you must issue the WITH EQUEL clause.

E_E60016 "The ADA statement %0c is not supported."

Explanation: Statements that modify the internal representation of variables that interact with Ingres are not supported.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application that uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Department information is stored in program variables. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.

- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo (the "toberesolved" database table, described below) to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second is for the Employee table. The **create** statements used to create the tables are shown below. The cursors retrieve all the information in their respective tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, both from the Department table and the Employee table, is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the EQUEL statements. The program illustrates table creation, multi-query transactions, all cursor statements and direct updates. For purposes of brevity, error handling on data manipulation statements is simply to close down the application.

The following two **create** statements describe the Employee and Department database tables:

```
##  create dept
##      (name       = c12,      ! Department name
##       totsales   = money,    ! Total sales
##       employees  = i2)       ! Number of employees

##  create employee
##      (name   = c20,          ! Employee name
##       age    = i1,           ! Employee age
##       idno   = i4,           ! Unique employee id
##       hired  = date,         ! Date of hire
##       dept   = c10,          ! Employee department
##       salary = money)        ! Yearly salary

10  !
    ! Program: Process_Expenses
    ! Purpose: Main entry of the application. Initialize the database,
    !          process each department, and terminate the session.
    !

    program Process_Expenses

        external byte function Init_Db

        print 'Entering application to process expenses.'
        if (Init_db = 1) then
            call Process_Depts
            call End_Db
                print 'Successful completion of application.'
        end if
```

```
    end program ! Process_Expenses

    !
    ! Function: Init_Db
    ! Purpose:  Initialize the database. Start up the database,
    !           and abort if an error. Before processing employees,
    !           create the table for employees who lose their
    !           department, "toberesolved". Initiate the
    !           multi-statement transaction.
    ! Returns:
    !           0 - Failed to start application.
    !           1 - Succeeded in starting application.
    !

20  ## function byte Init_Db

    ##      declare integer ing_error
        external integer Close_Down

    ##      ingres personnel

            print 'Creating "To_Be_Resolved" table.'
    ##      create toberesolved
    ##          (name = char(20),
    ##           age = smallint,
    ##           idno = integer,
    ##           hired = date,
    ##           dept = char(10),
    ##           salary = money)

    ##      inquire_ingres (ing_error = ERRORNO)
            if (ing_error > 0) then

                print 'Fatal error creating application table.'
                Init_Db = 0 ! Failed

            else

    ##           begin transaction
            !
            ! Inform Ingres runtime system about error handler
            ! All errors from here on close down the application.
            !
            call IIseterr(Close_Down)
            Init_Db = 1                 ! Ok

        end if

    ## end function                     ! Init_Db

        !
        ! Subroutine: End_Db
        ! Purpose:      Close off the multi-statement transaction and access
        !               to the database after successful completion of the
        !               application.
        !

30  ##  sub End_Db

    ##      end transaction
    ##      exit

    ## end sub                          ! End_Db

    !
```

```
    ! Subroutine: Process_Depts
    ! Purpose:    Scan through all the departments, processing
    !             each one. If the department has made less
    !             than $50,000 in sales, then the department
    !             is dissolved.
    !             For each department process all the
    !             employees (they may even be moved to another
    !             database table).
    !             If an employee was terminated, then update
    !             the department's employee counter.
    !

40 ## sub Process_Depts

        ! Dept_Rec corresponds to the "dept" table
    ##  record Dept_Rec
    ##      string dname = 12
    ##       double totsales
    ##       word employees
    ##  end record
    ##  declare Dept_Rec dpt

    ##  declare integer no_rows          ! Cursor loop control
    ##  declare integer emps_term        ! Employees terminated
        declare byte deleted_dept        ! Was the dept deleted?
        declare string dept_format       ! Formatting value
        declare byte dept_err

        ! Minimum sales of department
    ##  declare real constant MIN_DEPT_SALES = 50000.0

        no_rows = 0

    ##  range of d is dept
    ##  declare cursor deptcsr for
    ##      retrieve (d.name, d.totsales, d.employees)
    ##      for direct update of (name, employees)

    ##  open cursor deptcsr

        while (no_rows = 0)

    ##      retrieve cursor deptcsr
    ##          (dpt::dname, dpt::totsales, dpt::employees)

    ##      inquire_equel (no_rows = ENDQUERY)

            if (no_rows = 0) then

                ! Did department reach minimum sales ?
                if (dpt::totsales < MIN_DEPT_SALES) then
    ##                  delete cursor deptcsr
                        deleted_dept = 1
                        dept_format = ' -- DISSOLVED --'
                else
                        deleted_dept = 0
                        dept_format = ''
                end if

            ! Log what we have just done
            print 'Department: ' + trm$(dpt::dname) + &
                ', Total Sales: ';
            print using '$$####.##', dpt::totsales;
            print dept_format

            ! Now process each employee in the department
```

```
            call Process_Employees
                (dpt::dname, deleted_dept, emps_term)

            ! If some employees were terminated,
            ! record this fact
            if (emps_term > 0 and deleted_dept = 0) then
        ##      replace cursor deptcsr
        ##      (employees = dpt::employees - emps_term)
            end if

        end if                  ! If a row was retrieved

        next                    ! Continue with cursor loop

        ##       close cursor deptcsr

        ## end sub                  ! Process_Depts

        !
        ! Subroutine:  Process_Employees
        ! Purpose:     Scan through all the employees for a
        !              particular department. Based on given
        !              conditions the employee may be terminated, or
        !              given a salary reduction.
        !              1. If an employee was hired since 1985 then
        !                 the employee is terminated.
        !              2. If the employee's yearly salary is more
        !                 than the minimum company wage of $14,000
        !                 and the employee is not close to retirement
        !                 (over 58 years of age), then the employee
        !                 takes a 5% salary reduction.
        !              3. If the employee's department is dissolved
        !                 and the employee is not terminated, then
        !                 the employee is moved into the
        !                "toberesolved" table.
        ! Parameters:
        !              dept_name - Name of current department.
        !              deleted_dept - Is department dissolved?
        !              emps_term - Set locally to record how many
        !                             employees were terminated
        !                             for the current department.

    50 ##   sub Process_Employees(string dept_name,
        ##                        byte deleted_dept,
        ##                        integer emps_term)

            ! Emp_Rec corresponds to the "employee" table
        ##  record Emp_Rec
        ##      string      ename = 20
        ##      byte        age
        ##      integer     idno
        ##      string      hired = 25
        ##      real        salary
        ##      integer     hired_since_85
        ##  end record
        ##  declare Emp_Rec erec

            ! Minimum employee salary
        ##  declare real constant MIN_EMP_SALARY = 14000.00
        ##  declare byte constant NEARLY_RETIRED = 58
        ##  declare real constant SALARY_REDUC = 0.95

        ##  declare byte no_rows            ! For cursor loop control
            declare string title            ! Formatting values
            declare string description
```

```
    no_rows = 0
    emps_term = 0                       ! Initialize how many

    !
    !Note the use of the Ingres function to find out who was hired
    !since 1985.
    !
##      range of e is employee
##      declare cursor empcsr for
##      retrieve (e.name, e.age, e.idno, e.hired, e.salary,
##          res =
             int4(interval("days", e.hired - date("01-jan-1985"))))
##      where e.dept = dept_name
##      for direct update of (name, salary)

##  open cursor empcsr

    while (no_rows = 0)

##      retrieve cursor empcsr
##          (erec::ename, erec::age, erec::idno,
##           erec::hired, erec::salary, erec::hired_since_85)

## inquire_equel (no_rows = ENDQUERY)

    if (no_rows = 0) then

        ! Terminate if new employee
        if (erec::hired_since_85 > 0) then
##          delete cursor empcsr
            title = 'Terminated: '
            description = 'Reason: Since 85.'
            emps_term = emps_term + 1

        ! Reduce salary if not nearly retired
        else
            if (erec::salary > MIN_EMP_SALARY) then

            if (erec::age < NEARLY_RETIRED) then
##                  replace cursor empcsr
##                          (salary = salary * SALARY_REDUC)
                    title = 'Reduction: '
                    description = 'Reason: Salary.'
            else
                    ! Do not reduce salary
                    title = 'No Changes: '
                    description = 'Reason: Retiring.'
            end if

                    ! Else leave employee alone
            else
                    title = 'No Changes: '
                    description = 'Reason: Salary.'
            end if
        end if

        ! Was employee's department dissolved ?
        if (deleted_dept = 1) then

##          append to toberesolved (e.all)
##          where e.idno = erec::idno
##           delete cursor empcsr
        end if

        ! Log the employee's information
            print ' ' + title;
```

```
                print str$(erec::idno);
                print ', ' + trm$(erec::ename) + ', ';
                print str$(erec::age) + ', ';
                print using '$$####.##', erec::salary;
                print '; ' + description
            end if          ! If a row was retrieved

        next                ! Continue with cursor loop

        ##      close cursor empcsr

        ## end sub          ! Process_Employees

        !
        ! Function: Close_Down
        ! Purpose:  If an error occurs during the execution of an
        !           EQUEL statement this error handler is called.
        !           Errors are printed and the current database session
        !           is terminated. Any open transactions are implicitly
        !           closed.
        ! Parameters:
        !           ingerr - Integer containing Ingres error number.
        !
60      ## function integer Close_Down (integer ingerr)
        ##      declare string err_text

        ##      inquire_ingres (err_text = ERRORTEXT)
        ##      exit
                print 'Closing down because of database error: '
                print err_text
                stop                            ! Exit BASIC
                    Close_Down = ingerr
        ## end function                         ! Close_Down
```

The Employee Query Interactive Forms Application

This EQUEL/FORMS application uses a form in **query** mode to view a subset of the Employee table in the Personnel database. An Ingres query qualification is built at runtime using values entered in fields of the form "empform."

The objects used in this application are:

Object	Description
personnel	The program's database environment.
employee	A table in the database, with six columns: name (**c20**) age (**i1**) idno (**i4**) hired (**date**) dept (**c10**) salary (**money**)

Object	Description
empform	A VIFRED form with fields corresponding in name and type to the columns in the Employee database table. The Name and Idno fields are used to build the query and are the only updatable fields. "Empform" is a compiled form.

The application is driven by a **display** statement that allows the runtime user to enter values in the two fields that build the query. The Build_Query and Exec_Query procedures make up the core of the query that is run as a result. Note the way the values of the query operators determine the logic used to build the **where** clause in Build_Query. The **retrieve** statement encloses a **submenu** block that allows the user to step through the results of the query.

No updates are performed on the values retrieved, but any particular employee screen may be saved in a log file through the **printscreen** statement.

The following **create** statement describes the format of the Employee database table:

```
    ##  create employee
    ##      (name       = c20,          ! Employee name
    ##       age        = i1,           ! Employee age
    ##       idno       = i4,           ! Unique employee id
    ##       hired      = date,         ! Date of hire
    ##       dept       = c10,          ! Employee department
    ##       salary     = money)                ! Annual salary

10  ##   declare ingres
    ##   external integer empform

        ! Program: Employee_Query

        ! Initialize global WHERE clause qualification buffer
        ! to be an Ingres default qualification that is always true.
    ##   common(globs) string where_clause = 100
         where_clause = '1=1'

    ##   forms
    ##   message "Accessing Employee Query Application . . ."
    ##   ingres personnel

    ##   range of e is employee

    ##      addform empform

    ##      display #empform query
    ##      initialize

    ##      activate menuitem "Reset"
    ##      {
    ##                  clear field all
    ##      }

    ##      activate menuitem "Query"
    ##      {
                ! Verify validity of data
    ##          validate
```

```
                call Build_Query
                call Exec_Query
    ##      }

    ##      activate menuitem "LastQuery"
    ##      {
                        call Exec_Query
    ##       }

    ##      activate menuitem "End", frskey3
    ##      {
    ##                       breakdisplay
    ##      }
    ##      finalize

    ##      clear screen
    ##      endforms
    ##      exit

    ##      end                             ! main program

    ! Procedure: Build_Query
    ! Purpose:   Build an Ingres query from the values in the
    !            "name" and "idno" fields in "empform".
    ! Parameters:
    !                                       None

100 ##   sub Build_Query

            ! Global WHERE clause qualification buffer
    ##      common(globs) string where_clause = 100

    ##      declare string ename, integer eidno

            ! Query operator table that maps integer values to
            ! string query operators.
    ##      dim string operators(6)
            mat read operators
            data "= ", "!=", "< ", "> ", "<=", ">="

110         ! Operators corresponding to the two fields,
            ! that index into the "operators" table.
    ##      declare integer opername, operidno
    ##      getform #empform
    ##          (ename = name, opername = getoper(name),
    ##           eidno = idno, operidno = getoper(idno))

            ! Fill in the WHERE clause
            if (opername = 0% and operidno = 0%) then

                ! Default qualification
                where_clause = '1=1'

            else
                if (opername = 0% and operidno <> 0%) then

                        ! Query on the "idno" field
                        where_clause = 'e.idno' +                &
                        operators(operidno) + str$(eidno)

                else
                        if (opername <> 0% and operidno = 0%) then

                                ! Query on the "name" field
                                 where_clause = 'e.name' +       &
```

```
                                operators(opername) +            &
                                 '"' + ename + '"'

                        else ! (opername <> 0% and operidno <> 0%)

                                ! Query on both fields
                                where_clause = 'e.name' +        &
                                operators(opername) +            &
                                '"' + ename + '" and '           &
                                + 'e.idno' + &
                                operators(operidno) +            &
                                str$(eidno)
                        end if

                end if

            end if

    ##  end sub ! Build_Query

        ! Subroutine:  Exec_Query
        ! Purpose:     Given a query buffer, defining a WHERE clause
        !              issue a RETRIEVE to allow the runtime use to
        !              browse the employees found with the given
        !              qualification.
        ! Parameters:
        !              None

200 ##  sub Exec_Query

                ! Global WHERE clause qualification buffer
    ##          common(globs) string where_clause = 100

    ##          declare string           ename, ! Employee data
    ##          word                     eage,
    ##          integer                  eidno,
    ##          string                   ehired,
    ##          string                   edept,
    ##          real                     esalary

                declare byte rows ! Were rows found
                rows = 0%

                ! Issue query using WHERE clause
    ##          retrieve (
    ##                  ename = e.name, eage = e.age,
    ##                  eidno = e.idno, ehired = e.hired,
    ##                  edept = e.dept, esalary = e.salary)
    ##          where where_clause
    ## {
                        rows = 1%
                        ! Put values up and display them
    ##                  putform #empform (
    ##                  name = ename, age = eage,
    ##                  idno = eidno, hired = ehired,
    ##                  dept = edept, salary = esalary)
    ##                  redisplay

    ##                  submenu
    ##                  activate menuitem "Next" , frskey4
    ##                  {
                                ! Do nothing, and continue with the
                                ! RETRIEVE loop. The last one will
                                ! drop out.
    ##                  }
    ##                   activate menuitem "Save" , frskey8
```

```
##                  {
                            ! Save screen data in log file
##                           printscreen (file = "query.log")

                            ! Drop through to next employee
##                  }
##                  activate menuitem "End" , frskey3
##                   {
                            ! Terminate the RETRIEVE loop
##                           endretrieve
##                  }
##          }
            if (rows = 0%) then
##          message "No rows found for this query"
        else
##          clear field all
##          message "No more rows. Reset for next query"
        end if

##      sleep 2

## end sub                          ! Exec_Query
```

The Table Editor Table Field Application

This EQUEL/FORMS application uses a table field to edit the Person table in the Personnel database. It allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate their use and their interaction with an Ingres database.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**c20**) age (**i2**) number (**i4**) Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**c20**) age (**i4**) When initialized, the table field includes the hidden number (**i4**) column.

At the start of the application, a **retrieve** statement is issued to load the table field with data from the Person table. Once the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated or deleted. When finished, the values are unloaded from the table field, and, in a multi-query transaction, the user's updates are transferred back into the Person table.

The following **create** statement describes the format of the Person database table:

```
    ##  create person
    ##          (name    = c20,          ! Person name
    ##           age     = i2,           ! Age
    ##           number  = i4)           ! Unique id number
10  ##  declare ingres

        ! Person information corresponds to "person" table
    ##  declare string pname,            ! Full name
    ##          word p_age,              ! Age
    ##          integer pnumber,         ! Unique person number
    ##          integer persmaxid        ! Maximum person id number
        ! Table field row states
        declare byte constant ROW_UNDEF = 0      ! Empty or undefined row
        declare byte constant ROW_NEW = 1        ! Appended by user
        declare byte constant ROW_UNCHANGE=2     ! Prog loaded,not updated
        declare byte constant ROW_CHANGE = 3     ! Prog loaded and updated
        declare byte constant ROW_DELETE = 4     ! Deleted by program
        ! Table field entry information
    ##  declare integer state,                   ! State of data set row
    ##          recnum,                          ! Record number
    ##          lastrow                          ! Last row in table field

        ! Utility buffers
    ##  declare string  search,                  ! Name to find in loop
    ##          msgbuf,                          ! Message buffer
    ##          password,                        ! Password buffer
    ##          respbuf                          ! Response buffer

        ! Error handling variables for database updates
    ##  declare integer upd_err,                 ! Updates error
    ##          upd_rows                         ! Number of rows updated
        declare byte upd_commit                  ! Commit updates
        declare byte save_changes                ! (1 = true, 0 = false)
        ! Start up Ingre and the Ingre/Forms system
        ! We assume no Ingres errors will happen during screen updating.

    ##   ingres "personnel"

    ##  forms

    !   Verify that the user can edit the "person" table
    ##  prompt noecho ("Password for table editor: ", password)
        if (password <> "MASTER_OF_ALL") then
    ##          message "No permission for task. Exiting . . ."
    ##          endforms
    ##          exit
                goto endprog
        end if

    ##      message "Initializing Person Form . . ."
    ##      forminit personfrm

    ! Initialize "persontbl" table field with a data set in
    ! FILL mode so that the runtime user can append rows.
```

```
! To keep track of events occurring to original rows that will
! be loaded into the table field, hide the unique person number.

##  inittable personfrm persontbl fill (number = integer)
 ! Load the information from the "person" table into the
 ! person variables. Also save away the maximum person id number.

## message "Loading Person Information . . ."

## range of p is person

   ! Fetch data into person record, and load table field
## retrieve (pname = p.name, p_age = p.age, pnumber = p.number)
## {
## loadtable personfrm persontbl
## (name = pname, age = p_age, number = pnumber)
## }

   ! Fetch the maximum person id number for later use.
   ! PERFORMANCE NOTE: max() will do a sequential scan of table.

## retrieve (persmaxid = max(p.number))

   ! Display the form and allow runtime editing

## display personfrm update
## initialize

## ! Provide a menu, as well as the system FRS key to scroll
## ! to both extremes of the table field. Note that a comment
## ! between DISPLAY loop components MUST be marked with a ##.

## activate menuitem "Top" ,frskey5
## {
## scroll personfrm persontbl TO 1 ! Backward
## }

## activate menuitem "Bottom" , frskey6
## {
## scroll personfrm persontbl to end ! Forward
## }

## activate menuitem "Remove"
## {
   ! Remove the person in the row the user's cursor
   ! is on. If there are no persons, exit operation
   ! with message. Note that this check cannot
   ! really happen as there is always at least one
   ! UNDEFINED row in FILL mode.

##  inquire_frs table personfrm
##                  (lastrow = lastrow(persontbl))
        if (lastrow = 0%) then
##          message "Nobody to Remove"
##          sleep 2
##          resume field persontbl
        end if

##      deleterow personfrm persontbl ! Recorded for later
##  }

##   activate menuitem "Find" , frskey7
##  {
             ! Scroll user to the requested table field entry.
             ! Prompt the user for a name, and if one is typed
             ! in loop through the data set searching for it.
```

```
##          prompt ("Name of person: ", search)
            if (len(search) = 0%) then
##                  resume field persontbl
            end if

##          unloadtable personfrm persontbl
##                   (pname = name, recnum = _record,
##                     state = _state)
##          {
                     ! Do not compare with deleted rows
                    if (state <> ROW_DELETE and pname = search) then
##                        scroll personfrm persontbl to recnum
##                                resume field persontbl
                    end if
##          }

            ! Fell out of loop without finding name. Issue error.
            msgbuf = 'Person "' + search + &
                     '" not found in table [HIT RETURN] '
##          prompt noecho (msgbuf, respbuf)
##          }

##   activate menuitem "Save", frskey8
## {
##          validate field persontbl
            save_changes = 1
##          breakdisplay
## }

##  activate menuitem "Quit", frskey2
##  {
            save_changes = 0
##          breakdisplay
##  }
##   finalize

     if (save_changes = 0%) then
##          endforms
##          exit
            goto endprog
     end if

    ! Exit person table editor and unload the table field.
    ! If any updates, deletions or additions were made,
    ! duplicate these changes in the source table.
    ! If the user added new people we must assign a unique
    ! person id before returning it to the database table.
    ! To do this, we increment the previously saved
    ! maximum id number with each APPEND.

##  message "Exiting Person Application . . ."

    ! Do all the updates in a transaction (for simplicity,
    ! this transaction does not restart on DEADLOCK error: 4700)
##   begin transaction
        upd_commit = 1%

    ! Handle errors in the UNLOADTABLE loop, as we want to
    ! cleanly exit the loop, after cleaning up the transaction.
##  unloadtable personfrm persontbl
##          (pname = name, p_age = age,
##           pnumber = number, state = _state)
##  {
            select state
```

```
                    case = ROW_NEW

                    ! Filled by user. Insert with new unique id
                    persmaxid = persmaxid + 1%
##                  repeat append to person
##                          (name = @pname,
##                           age = @p_age,
##                           number = @persmaxid);

                    case = ROW_CHANGE

                    ! Updated by user. Reflect in table
##                  repeat replace p
##                          (name = @pname, age = @p_age)
##                          where p.number = @pnumber

                    case = ROW_DELETE

                    ! Deleted by user, so delete from table.
                    ! Note that only original rows are saved
                    ! by the program, and not rows appended
                    ! at runtime.
##                  repeat delete p where p.number = @pnumber

                case else

                    ! Else UNDEFINED or UNCHANGED
                    ! No updates required.
            end select

            ! Handle error conditions -
            ! If an error occurred, then abort the transaction.
            ! If a no rows were updated then inform user, and
            ! prompt for continuation.
##          inquire_equel (upd_err = errorno, upd_rows = rowcount)

            if (upd_err > 0%) then ! Abort on error
                    upd_commit = 0%
##                  message "Aborting updates . . ."
##                  abort
##                  endloop

            else
                    if (upd_rows = 0%) then ! May want to stop

                    msgbuf = 'Person "' + pname + &
                            '" not updated. Abort all updates?'
##                  prompt noecho (msgbuf, respbuf)
                            if (respbuf = "Y" or respbuf = "y") then
                                    upd_commit = 0%
##                                  abort
##                                  endloop
                            end if
                    end if
            end if
##  }

    if (upd_commit) then
##      end transaction ! Commit the updates
    end if

##  endforms ! Terminate the Forms and Ingres
##  exit

    endprog:
##  end
```

The Professor-Student Mixed Form Application

This EQUEL/FORMS application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**c25**) pdept (**c10**) See its **create** statement below for a full description.
student	A database table with seven columns: sname (**c25**) sage (**i1**) sbdate (**c25**) sgpa (**f4**) sidno (**i1**) scomment (text(**200**)) advisor (**c25**) See the **create** statement below for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has the Pname and Pdept fields, which correspond to the information in the Professor table, and table field studenttbl. The Pdept field is display-only. "Masterfrm" is a compiled form.
studenttbl	A table field in "masterfrm" with the sname and sage columns. When initialized, it also has five more hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the Sgpa, Scomment and Sadvisor fields are updatable. All other fields are display-only. "Studentfrm" is a compiled form.

Object	Description
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the "studentfrm" form and the studenttbl table field.

The program uses the "masterfrm" as the general-level master entry, in which data can only be retrieved and browsed, and the "studentfrm" as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the Pname (professor name) field and then selects the **Students** menu operation. The operation fills the displayed and hidden columns of the Studenttbl table field with detailed information of the students reporting to the named professor.

The user may then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student may be requested by selecting the **Zoom** menu operation. This operation displays the form "studentfrm." The fields of "studentfrm" are filled with values stored in the hidden columns of "studenttbl."

The user can make changes to the Sgpa, Scomment and Sadvisor fields. If validated, these changes are written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

The following two **create** statements describe the Professor and Student database tables:

```
    ##   create student                ! Graduate student table
    ##            (sname = c25,        ! Name
##             sage = i1,              ! Age
##             sbdate = c25,           ! Birth date
##             sgpa = f4,              ! Grade point average
##             sidno = i4,             ! Unique student number
##             scomment = text(200),   ! General comments
##             sadvisor = c25)   ! Advisor's name

    ##  create professor              ! Professor table
##           (pname = c25,            ! Professor's name
##            pdept = c10)            ! Department

10  ## declare ingres

     ! Master and student compiled forms (imported objects)
    ## external integer masterfrm, studentfrm

     ! Program: Prof_Student
     ! Purpose: Main body of "Professor Student" Master-Detail application.

    ! Graduate student record maps to "student" database table
     ## record Student_Rec
     ##           string sname = 25
     ##           byte sage
     ##           string sbdate = 15
     ##           real sgpa
     ##           integer sidno
```

```
##           string scomment = 200
##           string sadvisor = 25
## end record
## declare Student_Rec grad

! Professor record maps to "professor" database table
## record Prof_Rec
##           string pname = 25
##           string pdept = 10
## end record
## declare Prof_Rec prof

! Useful forms runtime information
## declare integer lastrow,   ! Lastrow in table field
##                 istable    ! Is a table field?

! Utility buffers
## declare string msgbuf,       ! Message buffer
##                respbuf,      ! Response buffer
##                old_advisor   ! Old advisor before ZOOM

! Function: Student_Info_Changed
! Purpose:  Allow the user to zoom into the details of a
!           selected student. Some of the data can be
!           updated by the user. If any updates were made,
!           then reflect these back into the database table.
!           The procedure returns 1 if any changes were made.
! Parameters:
!           None
! Returns:
!           1/0 - Changes were made to the database.
!           Sets the global "grad" record with the new data.
## def integer Student_Info_Changed

## declare integer changed,         ! Changes made to the form?
##                 valid_advisor   ! Valid advisor name?

! Display the detailed student information
## display #studentfrm fill
## initialize
##          (sname = grad::sname,
##                   sage = grad::sage,
##                   sbdate = grad::sbdate,
##                   sgpa = grad::sgpa,
##                   sidno = grad::sidno,
##                   scomment = grad::scomment,
##                   sadvisor = grad::sadvisor)

## activate menuitem "Write"
## {

If changes were made then update the
! database table. Only bother with the
! fields that are not read-only.

## inquire_frs form (changed = change)

if (changed = 1) then
##      validate
##      message "Writing to database. . ."

##      getform
##         (grad::sgpa = sgpa,
##          grad::scomment = scomment,
##          grad::sadvisor = sadvisor)
```

```
   ! Enforce integrity of professor name
     valid_advisor = 0
##         retrieve
##            (valid_advisor = 1)
##               where p.pname = grad::sadvisor

     if (valid_advisor = 0) then
##                 message "Not a valid advisor name"
##                 sleep 2
##                 resume field sadvisor
     else
##                 replace s
##                          (sgpa = grad::sgpa,
    ##                        comment = grad::scomment,
    ##                        sadvisor = grad::sadvisor)
    ##                        where s.sidno = grad::sidno
    ##             breakdisplay
     end if
    end if
## } ! "Write"

## activate menuitem "End" , frskey3
## {
         ! Quit without submitting changes
         changed = 0
##       breakdisplay
## } ! "Quit"
## finalize

Student_Info_Changed = changed

## end def ! Student_Info_Changed;

! Start up Ingres and the Forms system
## forms

## message "Initializing Student Administrator . . ."
## ingres personnel

## range of p is professor, s is student

## addform masterfrm
## addform studentfrm

! Initialize "studenttbl" with a data set in READ mode.
! Declare hidden columns for all the extra fields that
! the program will display when more information is
! requested about a student. Columns "sname" and "sage"
! are displayed, all other columns are hidden, to be
! used in the student information form.

## inittable #masterfrm studenttbl read
##          (sbdate = char(25),
##                  sgpa = float,
##                  sidno = integer,
##                  scomment = char(200),
##                  sadvisor = char(20))
! Drive the application, by running "masterfrm", and
! allowing the user to "zoom" into a selected student.
## display #masterfrm update

## initialize
## {
##          message "Enter an Advisor name . . ."
##          sleep 2
## }
```

```
## activate menuitem "Students", field "pname"
## {
        ! Load the students of the specified professor
##      getform (prof::pname = pname)

         ! If no professor name is given then resume
         if (len(edit$(prof::pname,8)) = 0) then
             ## resume field pname
         end if

         ! Verify that the professor exists. If not print
         ! print a message, and continue. We assume that
         ! each professor has exactly one department.

         prof::pdept = space$(10)
##        retrieve (prof::pdept = p.pdept)
##            where p.pname = prof::pname

         ! If no professor report error
         if (len(edit$(prof::pdept,8)) = 0) then
                    msgbuf = 'No professor with name "' &
                    + trm$(prof::pname) + '" [RETURN]'
##                  prompt noecho (msgbuf, respbuf)
##                  clear field all
##                  resume field pname
         end if

         ! Fill the department field and load students
##         message "Retrieving Student Information . . ."

##       putform (pdept = prof::pdept)
##       clear field studenttbl
##       redisplay ! Refresh for query

         ! With the advisor name, load into the "studenttbl" table field all
         ! the graduate students who report to the professor with that name.
         ! Columns "sname" and "sage" will be displayed,
         ! and all other columns will be hidden.

##       retrieve
##                  (grad::sname = s.sname,
##                   grad::sage = s.sage,
##                   grad::sbdate = s.sbdate,
##                   grad::sgpa = s.sgpa,
##                   grad::sidno = s.sidno,
##                   grad::scomment = s.scomment,
##                   grad::sadvisor = s.sadvisor)
##                   where s.sadvisor = prof::pname
##                  {
##                  loadtable #masterfrm studenttbl
##                          (sname = grad::sname,
##                           sage = grad::sage,
##                           sbdate = grad::sbdate,
##                           sgpa = grad::sgpa,
##                           sidno = grad::sidno,
##                           comment = grad::scomment,
##                           advisor = grad::sadvisor)
##                  }

##       resume field studenttbl
## } ! "Students"

## activate menuitem "Zoom"
## {
         ! Confirm that user is on "studenttbl", and that
```

```
        ! the table field is not empty. Collect data from
        ! the row and zoom for browsing and updating.

##      inquire_frs field #masterfrm (istable = table)
        if (istable = 0) then
##          prompt noecho
##                 ("Select from the student table [RETURN]",
##                  respbuf)
##          resume field studenttbl
         end if

    ## inquire_frs table #masterfrm (lastrow = lastrow)
    if (lastrow = 0) then
##          prompt noecho
##          ("There are no students [RETURN]",
##           respbuf)
##           resume field pname
    end if

    ! Collect all data on student into graduate record
##   getrow #masterfrm studenttbl
##         (grad::sname = sname,
##          grad::sage = sage,
##          grad::sbdate = sbdate,
##          grad::sgpa = sgpa,
##          grad::sidno = sidno,
##          grad::scomment = scomment,
##          grad::sadvisor = sadvisor)

    ! Display "studentfrm", and if any changes were made
    ! make the updates to the local table field row.
    ! Only make updates to the columns corresponding to
    ! writable fields in "studentfrm". If the student
    ! changed advisors, then delete this row from the
    ! display.

    old_advisor = grad::sadvisor
    if (Student_Info_Changed = 1) then
           if not (old_advisor = grad::sadvisor) then
##                  deleterow #masterfrm studenttbl
           else
##                 putrow #masterfrm studenttbl
##                 (sgpa = grad::sgpa,
##                  scomment = grad::scomment,
##                  sadvisor = grad::sadvisor)
            end if
     end if

## }               ! "Zoom"

## activate menuitem "Quit" , frskey2
## {
##          breakdisplay
## }               ! "Exit"

## finalize

## clear screen
## endforms
## exit

## end             ! main
```

Chapter 7: Embedded QUEL for Pascal

This chapter describes the use of EQUEL with the Pascal programming language.

Note: EQUEL/Pascal is supported in the VMS operating environment only.

EQUEL Statement Syntax for Pascal

This section describes the language-specific ground rules for embedding QUEL database and forms statements in a Pascal program. An EQUEL statement has the following general syntax:

*EQUEL_statement*

For information on QUEL statements, see the *QUEL Reference Guide*. For information on EQUEL/FORMS statements, see the *Forms-based Application Development Tools User Guide*.

The following sections describe how to use the various syntactical elements of EQUEL statements as implemented in Pascal.

Margin

There are no specified margins for EQUEL statements in Pascal. Always place the two number signs (**##**) in the first two positions of the line. The rest of the statement can begin anywhere else on the line.

Terminator

No statement terminator is required for EQUEL/Pascal statements. It is conventional not to use a statement terminator in EQUEL statements, although the Pascal statement terminator, the semicolon (\;), is allowed at the end of EQUEL statements. The preprocessor ignores it.

For example, the following two statements are equivalent:

```
##      sleep 1
```

and

```
##      sleep 1;
```

The terminating semicolon may be convenient when entering code directly from the terminal using the **-s** flag. For information on using the **-s** flag to test the syntax of a particular EQUEL statement, see Precompiling, Compiling, and Linking an EQUEL Program in this chapter.

EQUEL statements that are made up of a few other statements, such as a **display** loop, only allow a semicolon after the last statement. For example:

```
##  display empform             { no semicolon here }
##  initialize                  { no semicolon here }
##  activate menuitem 'Help'    { no semicolon here }
##  begin
##      message 'No help yet';  { semicolon allowed }
##      sleep 2;                { semicolon allowed }
##  end
##  finalize;                   { Semicolon allowed on last statement }
```

Variable declarations made visible to EQUEL observe the normal Pascal declaration syntax. Thus, variable declarations must be terminated in the normal way for Pascal, with a semicolon.

Line Continuation

There are no special line-continuation rules for EQUEL/Pascal. EQUEL statements can be broken between words and continued on any number of subsequent lines. An exception to this rule is that you cannot continue a statement between two words that are reserved when they appear together, such as **declare cursor**. For a list of double keywords, see the *QUEL Reference Guide*. Each continuation line must be started with ## characters. Blank lines are permitted between continuation lines.

If you want to continue a character-string constant across two lines, end the first line with a backslash character (\), and continue the string at the beginning of the next line. In this case, do not place ## characters at the beginning of the continuation lines.

For examples of string continuation, see String Literals in this chapter.

Comments

Two kinds of comments can appear in EQUEL programs, EQUEL comments and host language comments. Host language comments are passed through by the preprocessor. EQUEL comments are not. Therefore, source code comments you want to retain in preprocessor output should be entered as host language comments.

EQUEL comments appear on lines with two number signs (##) as the first two characters. Pascal comments are on lines without ##. In EQUEL/Pascal programs, comments can be delimited by:

```
/* and */
{ and }
(* and *)
```

The following restrictions apply to any comments in an EQUEL/Pascal program, whether intended as EQUEL comments or Pascal comments:

- If anything other than ## appears in the first two positions of a line of EQUEL source, the precompiler treats the line as host code and ignores it. The only exception to this is a string-continuation line. (For examples, see String Literals in this chapter.)
- Comments cannot appear in string constants. In this context, the intended comment will be interpreted as part of the string constant.
- In general, EQUEL comments are allowed in EQUEL statements wherever a space may legally occur. However, no comments can appear between two words that are reserved when they appear together, such as **declare cursor**. See the list of EQUEL reserved words in the *QUEL Reference Guide*.

The following additional restrictions apply to Pascal comments, only:

- Pascal comments cannot appear between component lines of EQUEL block-type statements. These include **retrieve**, **initialize**, **activate**, **unloadtable**, **formdata**, and **tabledata**, all of which have optional accompanying blocks delimited by open and close braces. Pascal comment lines must not appear between the statement and its block-opening delimiter.

  For example:

```
## retrieve (ename = employee.name)
            { illegal to put a host comment here! }
## begin
            { a host comment is perfectly legal here }
            writeln ('employee name ', ename);
## end
```

- Pascal comments cannot appear between the components of compound statements, in particular the **display** statement. It is illegal for a Pascal comment to appear between any two adjacent components of the **display** statement, including **display** itself and its accompanying **initialize**, **activate**, and **finalize** statements.

  For example:

```
## display empform
        { illegal to put a host comment here! }
## initialize (empname = 'Frisco McMullen')
        /* host comment illegal here! }
## activate menuitem 'clear'
## begin
```

```
        { host comment here is fine }
##     clear field all
## end
        { host comment illegal here! }
## activate menuitem 'end'
## begin
##     breakdisplay
## end
        { host comment illegal here! }
## finalize
```

The *QUEL Reference Guide* specifies these restrictions on a statement-by-statement basis.

n On the other hand, EQUEL comments are legal in the locations described in the previous paragraph, as well as wherever a host comment is legal. For example:

```
## retrieve (ename = employee.name)
## { this is an equel comment, legal in this location
##          and it can span multiple lines }
## begin
        writeln ('employee name ', ename);
## end
```

The EQUEL/Pascal comment can be either of the two standard Pascal comments, delimited by (* and *) or { and }, or the QUEL comment, delimited by /* and */. For example:

```
## message 'No permission ...' (* No user access *)
## sleep 2 { Let the user read it }
## message 'See the DBA for help'
             /* Only the DBA can help */
```

You cannot mix delimiters: a comment starting with /* must end with */ and not with *) or }.

Other EQUEL language preprocessors use braces ({ and }) as EQUEL block delimiters in statements such as **retrieve** and **unloadtable**. If you are converting EQUEL statements from another language into Pascal, be sure to change those block delimiters to the EQUEL/Pascal block delimiters, **begin** and **end**, so that the preprocessor does not treat them as Pascal comment delimiters.

String Literals

You can use either double quotes or single quotes to delimit string literals in EQUEL/Pascal, as long as the same delimiter is used at the beginning and the end of any one string literal.

Whichever quote mark you use, you can embed that quote mark as part of the literal itself by doubling it. For example:

```
##  append comments (field1 = 'a single'' quote')
```

To include the backslash character as part of a string, precede it with another backslash.

When continuing an EQUEL statement to another line in the middle of a string literal, use a backslash immediately prior to the end of the first line. In this case, the backslash and the following newline character are ignored by the preprocessor, so that the following line can continue both the string and any further components of the EQUEL statement. Any leading spaces on the next line are considered part of the string. For example, the following are legal EQUEL statements:

```
##  message 'Please correct errors found in updating\
    the database tables.'
##  append to employee (empname = "Freddie \
Mac", empnum = 222)
```

You cannot use the Pascal concatenation operator (**+)** to continue string literals over lines. If the preprocessor needs to continue a string literal on the next line, it will generate the correct Pascal code.

Block Delimiters

EQUEL block delimiters mark the beginning and end of the embedded block-structured statements. The **retrieve** loop and the forms statements **display**, **unloadtable**, **submenu**, **formdata**, and **tabledata** are examples of block-structured statements. The block delimiters to such statements are the keywords **begin** and **end**. For example:

```
##  retrieve (ename = emp.name)
##  begin
        writeln(ename);
##  end
```

Other EQUEL languages use braces to delimit the blocks. The EQUEL/Pascal preprocessor treats those delimiters as comment delimiters.

Pascal Variables and Data Types

This section describes how to declare and use Pascal program variables in EQUEL.

Variable and Type Declarations

The following sections describe Pascal variable and type declarations in EQUEL.

Declaring the EQUEL Runtime Routines

The EQUEL generated runtime routines can be declared to the compiler in either of two ways: with the EQUEL **declare** statement or with the Pascal **inherit** attribute, which accesses an environment file containing the routines. The runtime routines must be declared by one of these methods at the program level prior to any EQUEL statements.

The Declare Statement

EQUEL/Pascal is delivered with a Pascal include file that contains external procedure and function declarations for the EQUEL runtime library. The EQUEL **declare** statement generates a Pascal **%include** statement for this file in order to make these declarations visible to the Pascal compiler. This statement must appear in the program's declaration section.

```
##  program something;
##  var
##          row : integer;
##          name : packed array[1..20] of Char;
##  declare     {Include EQUEL procedures}
##  begin
        ...
##  end.
```

The Inherit Attribute

EQUEL/Pascal also comes with a Pascal environment file that has the same declarations in the compiled **include** file. By means of the VMS Pascal **inherit** attribute, you can use this environment file instead of issuing a **declare** statement. Compilation should be slightly faster with this technique. The syntax for inheriting the EQUEL runtime routines is:

[inherit('EQUEL')] *program_heading;*

```
##  [inherit('EQUEL')] program test( input, output );
##  var
##          msg : varying[100] of Char;
##  begin
##          forms
            msg := 'No DECLARE statement was issued';
##          message msg
##  end.
```

For information on installing the environment file, see Precompiling, Compiling, and Linking an EQUEL Program in this chapter.

Declaring Types and Variables to EQUEL

EQUEL statements use Pascal variables to transfer data from a database or a form into the program and conversely. You must declare Pascal variables to EQUEL before using them in EQUEL statements. Pascal variables are declared to EQUEL by preceding the declaration with the ## mark. The declaration must be in a position syntactically correct for the Pascal language. Similarly, constants, types, and formal parameters used in EQUEL must be made known to EQUEL by preceding their declarations with the ## mark.

In general, each declared object can be referred to in the scope of the enclosing compilation unit. An object name cannot be redeclared in the same compilation unit scope. For details on the scope of types and variables, see Compilation Units and the Scope of Objects in this chapter.

Reserved Words in Declarations and Program Units

All EQUEL keywords are reserved; therefore, you cannot declare variables with the same names as those keywords. In addition, the following Pascal words, used in declarations and program units, are reserved and cannot be used elsewhere, except in quoted string literals:

array	**case**	**const**	**declare**	**do**
extern	**external**	**file**	**fortran**	**forward**
function	**label**	**module**	**otherwise**	**packed**
procedure	**program**	**record**	**type**	**var**
varying				

The word **module** cannot be used as an identifier in EQUEL/Pascal, although it is allowed in Pascal. The EQUEL preprocessor does not distinguish between uppercase and lower case in keywords.

Data Types and Constants

The EQUEL/Pascal preprocessor accepts the data types in the following table. The types are mapped to their corresponding Ingres type categories. For exact type mapping, see Data Type Conversion in this chapter.

Pascal Data Types and Corresponding Ingres Types

Pascal Type	Ingres Type
boolean	**integer**
integer	**integer**

Pascal Type	Ingres Type
unsigned	**integer**
real	**float**
single	**float**
double	**float**
char	**character**
indicator	**indicator**

Your program should not redefine any of the above types.

The following table maps the Pascal constants to their corresponding Ingres type categories.

Pascal Constants and Corresponding Ingres Types

Pascal Constant	Ingres Type
maxint	**integer**
true	**integer**
false	**integer**

The Integer Data Types

Several Pascal types are considered as integer types by the preprocessor as shown in the following table.

Pascal Integer Types

Description	Example
integer	Integer
4-byte subrange of integer	1..127
2-byte subrange of integer	[word] 0..32767
1-byte subrange of integer	[byte] 0..63
enumeration	(red, blue, green)
boolean	Boolean

All **integer** types are accepted by the preprocessor. Even though some integer types have Pascal constraints, such as the subranges and enumerations, EQUEL does not check these constraints, either during preprocessing or at runtime.

The type **boolean** is handled as a special type of **integer**. EQUEL treats the **boolean** type as an enumerated type and generates the correct code in order to use this type to interact with an Ingres integer. Enumerated types are described in more detail later.

The Indicator Type

An *indicator type* is a 2-byte integer type. There are three possible ways to use these in an application:

- In a statement that retrieves data from Ingres, you can use an indicator type to determine if its associated host variable was assigned a null.
- In a statement that sets data to Ingres, you can use an indicator type to assign a null to the database column, form field, or table field column.
- In a statement that retrieves character data from Ingres, you can use the indicator type as a check that the associated host variable is large enough to hold the full length of the returned character string.

EQUEL/Pascal predefines the 2-byte integer type **indicator**. As with other types, you should not redefine the **indicator** type. This type definition is in the file that is included when preprocessing the EQUEL **declare** or **inherit** directives. The type declaration syntax is:

```
type
        Indicator = [word] -32768..32767;
```

Because the type definition is in the referenced **include** file, you can only declare variables of type **indicator** after you have issued **declare** or **inherit**. This declaration does not preclude you from declaring indicator types of other 2-byte integer types.

The Floating-point Data Types

There are three floating-point types that are accepted by the preprocessor. The types **single** and **real** are the 4-byte floating-point types. The type **double** is the 8-byte floating-point type. Note that, although the preprocessor accepts **quadruple** data type declarations, it does not accept references to variables of type **quadruple**. (For more information on record types, see the Record Type Definition in this chapter.)

The Double Storage Format

EQUEL requires that the storage representation for **double** variables be **d_float**, because the EQUEL runtime system uses that format for floating-point conversions. If your EQUEL program has **double** variables that interact with the EQUEL runtime system, you must make sure they are stored in the **d_float** format. Because the default Pascal format is **d_float**, your program will automatically use the correct storage representation unless you use the **g_floating** compiler option. Any module compiled with this option must not use **double** variables or **float literals** to interact with Ingres. **Float literals** are treated as double precision numbers by Ingres. Note that EQUEL recognizes only **single**, and not **double** or **quadruple**, exponential notation for real constants. Thus, any real constants passed to Ingres are always single precision and are unaffected by the **g_floating** compiler option.

The Character Data Types

Three Pascal data types are compatible with Ingres string objects: **char**, **packed array of char**, and **varying of char**. Note that literal string constants are of type **packed array of char**. EQUEL allows only regular Pascal string literals: sequences of printing characters enclosed in single quotes. The VMS Pascal extensions of parenthesized string constructors and of nonprinting characters represented by their ASCII values in parentheses are not allowed.

The **char** data type does have some restrictions. Because of the mechanism used to pass string-valued arguments to the EQUEL runtime library, you cannot use a member of a **packed array of char** or **varying of char** to interact with Ingres. Also plain **array of char** (for example, not **packed** or **varying**) is not compatible with Ingres string objects; an element of such an array, however, is a **char** and as such is compatible.

For example, given the following legal declarations:

```
##  type
##      Alpha = 'a'..'z'; {1 character}
##      Packed_6 = packed array[1..6] of Char;
##           {6-char string}
##      Vary_6 = varying[6] of Alpha; {6-char string}
##      Array_6 = array[1..6] of Char;
##           {1-dimensional array}
##  var
##      letter: Alpha; {1 character}
##      p_str_arr: array[1..5] of Packed_6;
##           {Array of strings}
##      chr_arr: array[1..6] of Char;
##           {1-dimensional array}
##      two_arr: array[1..5] of Array_6;
##           {2-dimensional array of char}

##  v_string : Vary_6; {String}
```

these usages are legal:

```
##  message letter              {a char is a string}
```

```
##  message chr_arr[3]          {a char is a string}
##  message two_arr[2][5]       {a char is a string}
##  message v_string            {a varying array is a string}
##  message p_str_arr[2]        {a packed array is a string}
```

but these usages are illegal:

```
##  message chr_arr         {an array of chars is not a string}
##  message v_string[2]     {cannot index a varying array}
##  message p_str_arr[2][3] {Cannot index a packed array}
```

Declaration Syntax

The following sections describe the declaration syntax.

Attributes

In type definitions, EQUEL allows VMS Pascal attributes both at the beginning of the definition and just before the type name. The only attributes the preprocessor recognizes in type definitions are **byte**, **word**, and **long**. Any optional storage unit constant "(*n*)" appearing with the attribute is ignored by the preprocessor. The preprocessor also ignores all other attributes, although it allows them.

The following example shows how to use the **byte** attribute in order to convert a 4-byte integer subrange into a 1-byte variable.

```
##  var
##  v_i1 : [byte] -128..127;
```

Note that Pascal requires that a size attribute be at least as large as the size of its type. Therefore, the following declaration would be illegal, because 400 will not fit into one byte:

```
##  var
##  v_i1 : [byte] 0..400;
```

EQUEL/Pascal does not allow explicit attribute size conflicts, as, for example:

```
##  type
##      i1 = [byte] -128..127;
##  var
##      v_i2 : [word] i1;
```

In addition to appearing in type definitions, attributes can also precede a compilation unit, where they are ignored by the preprocessor, with the exception of the attribute "[inherit('EQUEL')]", which has the same effect as an EQUEL **declare** statement in the declaration section of the compilation unit. The **inherit** attribute should appear alone, because the preprocessor discards any attributes that appear with it. For more information using this attribute in EQUEL, see The Inherit Attribute in this chapter.

Label Declarations

EQUEL/Pascal no longer requires the use of EQUEL **label** declarations, required in earlier versions. As a better alternative, you should place the EQUEL ## mark before the header of each EQUEL compilation unit (**program**, **module**, **procedure**, or **function**) and the opening **begin** and closing **end** statements. If you do not either use the **label** declaration or mark the compilation unit header, you will get an error message if the preprocessor needs to generate labels, and the resulting code will not compile. For more information on compilation unit syntax, see Compilation Units and the Scope of Objects in this chapter.

The Syntax of Label Declarations

Earlier versions of EQUEL/Pascal allowed the declaration of program-declared labels without a terminating semicolon:

```
##  label
##      start, stop
```

EQUEL/Pascal still allows this syntax but generates a warning. You can avoid the warning by terminating the label with a semicolon:

```
##  label
##      start, stop;
```

You need not use a semicolon if you do not declare any labels yourself:

```
##  label
```

Constant Declarations

The syntax for a constant declaration is:

const *constant_name* = *constant_expr;*
{*constant_name* **= *constant_expr;***}

where a *constant_expr* is one of the following:

[**+**|**-**] *constant_number*
[**+**|**-**] *constant_name*
string_constant

Constants can be used to set Ingres values but cannot be assigned values from Ingres.

Syntax Notes:

1. A *constant_name* must be a legal Pascal identifier beginning with an alphabetic character or an underscore.

2. A *constant_number* can be either an integer or real number. It cannot be a numeric expression.
3. EQUEL/Pascal recognizes only **single**, and not **double** or **quadruple**, exponential notation for constants of type **real**.
4. The type of a *constant_name* is determined from the type of its *constant_expr*.
5. If a *constant_name* used as a *constant_expr* is preceded by a '+' or '-', it must be numeric.
6. EQUEL/Pascal does not support the declaration of arbitrary constant expressions.

```
##  const
##      min_sal     = 15000.00;     {Real}
##      pi          = 3.14159;      {Real}
##      max_emps    = +99;          {Integer}
##      max_credit  = 100000.00;    {Real}
##      max_debt    = -max_credit;  {Real}
##      yes         = 'y';          {Char}
```

Type Declarations

An EQUEL/Pascal type declaration has the following syntax:

type *type_name* = *type_definition;*
{*type_name* **= *type_definition;***}

where *type_definition* is any in the following table.

Type Definitions

Syntax	Category
type_name	renaming
(enum_identifier {,enum_identifier})	enumeration
[+\-] *constant .. [+|-]constant*	numeric or character subrange
^*type_name*	pointer
varying [*upper_bound*] **of** *char_type_name*	varying length string
[packed] array [*dimensions*] **of** *type_definition*	array
record *field_list* **end**	record
file of *type_definition*	file
set of *type_definition*	set

Each of these type definitions is discussed in its own section below. All type names must be legal Pascal identifiers beginning with an alphabetic character or an underscore.

Renaming Type Definition

The declaration for the renaming of a type uses the following syntax:

type *new_type_name* = *type_name;*

Syntax Notes:

1. The *type_name* must be either an EQUEL/Pascal type or a type name already declared to EQUEL such as **integer** or **real**.
2. The *new_type_name* cannot be **integer**, **real**, or **char**, or any other type listed at the beginning of this section.

```
##  type
##  NaturalInt = Integer; {A "natural" sized integer}
```

Enumeration Type Definition

The declaration for an enumeration type definition has the following syntax:

type *type_name* = *(enum_identifier* {**, *enum_identifier***} **);**

Syntax Notes:

1. An *enum_identifier* must be a legal Pascal identifier beginning with an alphabetic character or underscore.
2. The *enum_identifiers* are treated as 4-byte integer constant identifiers.
3. The *type_name* maps to a 1-byte integer if there are fewer than 257 enumerated identifiers. Otherwise, it maps to a 2-byte integer.
4. When an enumerated identifier is used as a value in an EQUEL statement, only the ordinal position of the identifier in the original enumerated list is important. In assigning a value to a variable of enumeration type, EQUEL passes the variable by address and assumes that the value is a legal one for the variable.

The following is an example of an enumeration type definition:

```
##  type
##      Table_Field_States =
##      (UNDEFINED, NEWROW, UNCHANGED, CHANGED, DELETED);
```

Subrange Type Definition

The syntax for declaring a subrange type definition is either:

type *type_name* = [**+**|**-**]*integer_const* **..** [**+**|**-**]*integer_const;*

or

type *type_name* = *string_const .. string_const;*

Syntax Notes:

1. An *integer_const* may be either an integer literal or a named integer constant.
2. A *string_const* must be either a string literal or the name of a string constant. Although the preprocessor accepts any length string constant, the compiler requires the constant to be a single character.

```
##  type
##      Alpha = 'a' .. 'z';
##      Months = 1 .. 12;
##      MinMax = -Value .. Value; {"Value" is an
##          integer constant}
##      Updated_States = CHANGED .. DELETED;
```

Pointer Type Definition

The declaration for a pointer type definition has the following syntax:

type *pointer_name* = ^*type_name;*

Syntax Note:

The *type_name* can be either a previously defined type or a type not yet defined. If the type has not yet been defined, the pointer type definition is a *forward pointer* definition. In that case, EQUEL requires that the *type_name* be defined before a variable of type *pointer_name* is used in an EQUEL statement.

The following example illustrates the use of the pointer type definition:

```
##  type
##      EmpPtr =^EmpRecord;
##              {Forward pointer declaration}
##      EmpRecord = record
##                  e_name : varying[40] of Char;
##                  e_salary : Real;
##                  e_id : Integer;
##                  e_next : EmpPtr;
##      end;

##  var
##      empnode = EmpPtr;
                ...

##      retrieve (empnode^ .e_ename = emp.name,
```

```
##        empnode^.e_salary = emp.salary,
##        empnode^.e_id = emp.id)
```

Varying Length String Type Definition

The declaration for a varying length string type definition has the following syntax:

type ***v****arying_type_name* = ***varying*** [*upper_bound*] *of char_type_name;*

Syntax Notes:

1. The *upper_bound* of a varying length string specification is not parsed by the EQUEL preprocessor. Consequently, an illegal upper bound (such as a non-numeric expression) will be accepted by the preprocessor but will later cause Pascal compiler errors. For example, both of the following type declarations are accepted, even though only the first is legal in Pascal:

   ```
   ##  type
   ##      String20 = varying[20] of Char;
   ##      What         = varying['upperbound'] of Char;
   ```

2. EQUEL/Pascal treats a variable of type **varying of char** as a string, not an array.

   ```
   ##  type
   ##           Pname = varying[100] of Char;
   ##  var
   ##           user_name : Pname;
       ...
   ##  append to person (name = user_name)
   ```

Array Type Definition

The declaration for an array type definition has the following syntax:

type *type_name* **=** [**packed**] **array** [*dimensions*] **of** *type_definition;*

Syntax Notes:

1. The *dimensions* of an array specification are not parsed by the EQUEL preprocessor. Consequently, an illegal dimension (such as a non-numeric expression) will be accepted by the preprocessor but will later cause Pascal compiler errors. For example, both of the following type declarations are accepted, even though only the first is legal in Pascal:

   ```
   ##  type
   ##      Square = array[1..10, 1..10] of Integer;
   ##      What = array['dimensions'] of Real;
   ```

   The preprocessor only verifies that an array variable is followed by brackets when used (except **packed array of char**—see below).

2. EQUEL/Pascal treats a variable of type **packed array of char** as a string, not an array. Therefore, it is not followed by brackets when used.
3. Components of a **packed array** cannot be passed to the EQUEL runtime routines. Therefore, you should not declare **packed arrays** to EQUEL, except for **packed arrays of char**, which are passed as a whole (for example, as character strings).

The following example illustrates the use of the array type definition:

```
##  type
##      Ssid = packed array [1..9] of Char;
##  var
##      user_ssid : Ssid;
    ...
##  append to person (ssno = user_ssid)
```

Record Type Definition

The declaration for a record type definition has the following syntax:

type *record_type_name* =
    record
        field_list [**;**]
    end;

where *field_list* is:

field_element {**;** *field_element*}
[**case** [*tag_name* ***:***] *type_name* ***of***
    [*case_element* {**;** *case_element*}]
    [**otherwise (*field_list*)**]]

where *field_element* is:

field_name {, *field_name*} : *type_definition*

and *case_element* is:

case_label {, *case_label*} : (*field_list*)

Syntax Notes:

1. All clauses of a record component have the same rules and restrictions as they do in a regular type declaration. For example, as with regular declarations, the preprocessor does not check dimensions for correctness.
2. In the **case** list, the *case_labels* may be numbers or names. The names need not be known to EQUEL.

3. Pascal host code is not a legal EQUEL **record** component. Consequently, all components of the record must be preceded by the ## mark. To minimize the effect of this restriction, the types **quadruple** and **set of** are allowed as legal types in an EQUEL record declaration. It is, however, an error to use variables of those types in EQUEL statements.
4. Components of a **packed** record cannot be passed to the EQUEL runtime routines. Therefore, you should not declare **packed** records to EQUEL.

The following example illustrates the use of a record type definition:

```
##  type
##          AddressRec = record
##              street: packed array[1..30] of Char;
##              town: packed array[1..10] of Char;
##              zip: 1 .. 9999;
##          end;

##          EmployeeRec = record
##              name: packed array[1..20] of Char;
##              age: [byte] 0 .. 128;
##              salary: Real;
##              address: AddressRec;
##              checked: Boolean;
##              scale: Quadruple;
                {Requires ##, but cannot be used by EQUEL}
##  end;
```

File Type Definition

The declaration for a file type definition has the following syntax:

type *type_name* = *file of* *type_definition;*

Syntax Notes:

1. A variable of type **file** can only be used with EQUEL through the file buffer. A file buffer for a given *type_definition* is referenced in the same manner as a pointer to the same type.
2. Components of a **packed** file cannot be passed to the EQUEL runtime routines. Therefore, you should not declare **packed** files to EQUEL.

The following example illustrates the use of a file type definition:

```
##  var
##          myfile : file of Integer;
##  begin
        ...

        get (myfile);
##      append to emp (floor = myfile^);
        ...

##      retrieve (myfile^ = emp.floor)
##      begin
                put (myfile);
##      end;
```

Set Type Definition

The declaration for a set type definition has the following syntax:

type *type_name* = ***set of t****ype_definition;*

Syntax Note:

1. Although set definitions are accepted by the preprocessor, no set variables can be used in EQUEL statements. As discussed in the section above on **record** declarations, **set** declarations are accepted only because all record components must be declared to EQUEL.

Variable Declarations

An EQUEL/Pascal variable declaration has the following syntax:

var *var_name* {**, *var_name***} **: *type_definition*** [**:= *initial_value***];
{*var_name* {**, *var_name***} **: *type_definition*** [**:= *initial_value***];}

Syntax Notes:

1. See the previous section for information on the *type_definition*.
2. The *initial_value* is not parsed by the preprocessor. Consequently, any initial value is accepted, even if it later causes a Pascal compiler error. Furthermore, the preprocessor accepts an initial value with any variable declaration, even where not allowed by the compiler. For example, both of the following initializations are accepted, even though only the first is legal in Pascal:

```
##  var
##          rowcount: Integer := 1;
##          msgbuf: packed array[1..100] of Char := 2;
```

The following is an example of a variable declaration:

```
##  var
##          rows, records:      0..500 := 0;
##          was_error:          Boolean;
##          msgbuf:             varying[100] of Char := ' ';
##          operators:  array[1..6] of packed array[1..2] :=
##                      ('= ', '!=', '< ', '> ', '<=', '>=');
##          employees : array[1..100] of EmployeeRec;
##          emp_ptr :           ^EmployeeRec;
##          work_days :         (MON, TUE, WED, THU, FRI);
##          day_name :          varying[8] of Char;
##          random_ints         file of Integer;
##          null_ind:           Indicator;
```

Formal Parameter Declarations

Most VMS Pascal formal parameter declarations are acceptable to EQUEL. Declared formal parameters are treated as local variables by EQUEL. Note that host code is not allowed in an EQUEL formal parameter section; therefore, all formal parameters to a procedure or function known to EQUEL must be preceded by the ## mark.

An EQUEL/Pascal formal parameter declaration has the following syntax:

formal_param_section {**; *formal_param_section***}

where *formal_param_section* is:

formal_var | *formal_routine* [**:=** [***%mechanism***] *default_value*]

A *formal_var* has the syntax:

[**var** | ***%mechanism***] *identifier* {***, identifier***} **: *typename_or_schema***

where *typename_or_schema* is one of the following:

type_name
varying [*upper_bound_identifier*] **of *type_name***
packed array [*schema_dimensions*] of *typename_or_schema*
array [*schema_dimensions* {**; *schema_dimensions***}] **of**
typename_or_schema

where *schema_dimensions* is:

lower_bound_identifier ***.. upper_bound_identifier : scalar_type_name***

A *formal_routine* has the syntax:

[***%mechanism***] *routine_header*

where *routine_header* is one of the following:

procedure *proc_name (* [*formal_parameter_declaration*] **)**
function *func_name (* [*formal_parameter_declaration*] **) :**
return_type_name

In a subprogram declaration, the syntax of a formal parameter declaration is:

procedure *proc_name (formal_parameter_declaration);*
...

or:

function *func_name (formal_parameter_declaration) : return_type_name;*
...

Syntax Notes:

1. The EQUEL preprocessor ignores the names of procedures and functions used as formal parameters, but checks their formal parameters for legality.
2. The *default_value* is not parsed by the preprocessor. Consequently, any default value is accepted, even if it later causes a Pascal compiler error. For example, both of the following parameter default values are accepted, even though only the first is legal in Pascal:

```
##  procedure Load_Table
##      (clear_it: Boolean := TRUE;
##        var is_error: Boolean := 'FALSE');
            ...
```

3. Any *mechanism* specification is ignored.
4. The scope of the parameters is the procedure or function in which they are declared. For detailed scope information, see Compilation Units and the Scope of Objects in this chapter.

The following example contains formal parameter declarations:

```
##  function GetEquelError( buf:varying[ub] of Char )
##  : Boolean;

##  procedure HandleError( procedure errorHandle(err : Integer);
##   var errNum : Integer );

##  function DoAppend( emp_id, floor : Integer;
##   name : varying[ub] of Char;
##   salary : Real ) : Integer;
```

Assembling and Declaring External Compiled Forms

You can pre-compile your forms in the Visual Forms Editor (VIFRED). This saves the time otherwise required at runtime to extract the form's definition from the database forms catalogs. When you compile a form in VIFRED, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED prompts you for the name of the file with the MACRO description. After the file is created, use the following command to assemble it into a linkable object module with the VMS command:

macro *filename*

This command produces an object file containing a global symbol with the same name as your form. Before the EQUEL/FORMS statement **addform** can refer to this global object, it must be declared in an EQUEL declaration section. The Pascal compiler requires that this be an *external* declaration. The syntax for a compiled form declaration is:

```
##      var
##      formname: [external] Integer;
```

Syntax Notes:

1. The *formname* is the actual name of the form. VIFRED gives this name to the address of the external object. The *formname* is also used as the title of the form in other EQUEL/FORMS statements. In all statements other than **addform** that use *formname* as an argument you must differentiate the name with a # sign.
2. The **external** attribute associates the object with the external form definition.

The example below shows a typical form declaration and illustrates the difference between using the form's object definition and the form's name.

```
##  var
##      empform: [external] Integer;
    ...

##  addform empform      {the global object}
##  display #empform     {the name of the form}
    ...
```

Concluding Example

The following example demonstrates some simple EQUEL/Pascal declarations:

```
##  program Concluding_Example( input, output );

##  const
##          MAX_PERSONS = 1000;
##  type
##          ShortShortInteger = [byte] -128 .. 127;
##          ShortInteger = [word] -32768 .. 32767;
                           {same as Indicator type}
##          String9 = packed array[1..9] of Char;
##          String12 = packed array[1..12] of Char;
##          String20 = packed array[1..20] of Char;
##          String30 = packed array[1..30] of Char;
##          VarString = varying[40] of Char;

##  record Datatypes_Rec = {Structure of all types}
##          d_byte : ShortShortInteger;
##          d_word : ShortInteger;
##          d_long : Integer;
##          d_single : Real;
##          d_double : Double;
##          d_string : String20;
##      end;
```

```
##  record Persontype_Rec = {Variant record}
##           age : ShortShortInteger;
##           flags : Integer;
##           case married : Boolean of
##               TRUE : (spouse_name : String30);
##               FALSE : (dog_name : String12);
##       end;
##  var
##       empform, deptform : [external] Integer; {Compiled forms}
##  dbname : String9;
##  formname, tablename, columnname : String12;
##  salary : Real;
##  d_rec : Datatypes_Rec;
##  person : Persontype_Rec;
##  person_store : array[1..MAX_PERSONS]
##  of Persontype_Rec;
##  person_null: array[1..10] of Indicator;

##  begin
        dbname := 'personnel';
        ...

##      end. {Concluding_Example}
```

Compilation Units and the Scope of Objects

Following Pascal conventions, all objects in an EQUEL/Pascal program are local to the scope in which they are declared and are visible in any nested scopes unless hidden by an intermediate redeclaration.

Constant, label, type, and variable names are local to the closest enclosing Pascal compilation unit. EQUEL/Pascal compilation units include programs, modules, procedures, and functions. The objects visible in their scopes include objects that are visible in the parent scope, formal parameters (if applicable), and local declarations. As in Pascal, once the preprocessor has exited the scope, the variables are no longer visible and cannot be referenced.

Note that compilation units that use EQUEL statements must be declared to EQUEL. This is accomplished by preceding the unit's header and its **begin** and **end** statements with the ## mark.

EQUEL does not support Pascal inherited environments, except for the special case of the EQUEL environment. For more information, see The Inherit Attribute in this chapter.

Predeclared Identifiers

EQUEL predeclares all the standard Pascal types and constants, which are listed in the section titled "Data Types and Constants," in a scope enclosing the entire program. You should not redefine any of these identifiers because the runtime library expects the standard definitions.

Compilation Unit Syntax

The following sections describe the compilation unit syntax.

The Program Unit

The syntax for an EQUEL/Pascal program definition is:

program *program_name [(identifier {, identifier})];*
[*declarations*]
begin
[*statements*]
end.

where *declarations* can include any of the following:

label *label_declarations*
const *constant_declarations*
type *type_declarations*
var *variable_declarations*
procedures
functions
host_code

For a detailed description of the various types of declarations, see Declaration Syntax in this chapter.

Syntax Notes:

1. The *program_name* and the *identifiers* are not processed by EQUEL.
2. The various declaration sections can appear in any order and can be repeated.
3. The **label** declaration section is allowed only for compatibility with earlier versions of Ingres.

The Module Unit

The syntax for an EQUEL/Pascal module definition is:

module *module_name [(identifier {, identifier})];*
[*declarations*]
end.

where *declarations* are the same as those for program units (see above). For a detailed description of the various types of declarations, see Declaration Syntax in this chapter.

Syntax Notes:

1. The *module_name* and the *identifiers* are not processed by EQUEL.
2. The various declaration sections can appear in any order and can be repeated.

```
##  module ExternalVars;
##  var
##      CurFormName, CurFieldName, CurColName :
##          varying[12] of Char;
##      CurTableRow : Integer;
##  end.
```

The Procedure

The syntax for an EQUEL/Pascal procedure is:

procedure *procedure_name [(formal_parameters)];*
    [*declarations*]
begin
    [*statements*]
end;

Syntax Notes:

1. The *procedure_name* is not processed by EQUEL.
2. Formal parameters and variables declared in a procedure are visible to the procedure and to any nested blocks.
3. For a description of formal parameters and their syntax, see Formal Parameter Declarations in this chapter.

```
##   procedure AppendRow( name : varying[ub] of Char;
##   age : Integer;
##   salary : Real );
##   begin
##    APPEND TO emp (#name = name, #age = age,
##      salary = salary)
##   end;
```

The Function

The syntax for an EQUEL/Pascal function is:

function *function_name [(formal_parameters)] : return_type_name*
    [*declarations*]
begin
    [*statements*]
end;

Syntax Notes:

1. The *function_name* is not processed by EQUEL.
2. Formal parameters and variables declared in a function are visible to the function and to any nested blocks.
3. For a description of formal parameters and their syntax, see Formal Parameter Declarations in this chapter.
4. EQUEL does not allow function calls to replace variables in executable statements. Therefore, EQUEL need not know the *return_type_name*.

The following is an example of a function:

```
##  function WasError( errorBuf : varying[ub] of
##              Char ) : Boolean;
##  const
##           EquelNoError = 0;
##  var
##           errNum : Integer;
##  begin
##      INQUIRE_EQUEL (errNum = error)
        if errNum = EquelNoError then
        begin
        errorBuf := ' ';
            WasError := FALSE;
        end else
        begin
            SetErrorBuf( errNum, errorBuf );
            WasError := TRUE;
        end;
##  end;
```

The Scope of Objects

As mentioned above, constants, variables and types are visible in the block in which they are declared. Objects can be redeclared only in a nested scope, such as in a nested procedure, but not in the same scope.

Note that you can declare record components with the same name if they are in different record types. The following example declares two records, each of which has the components "firstname" and "lastname":

```
##  type
##      Child = record
##          firstname: varying[20] of Char;
##          lastname: varying[20] of Char;
##          age: Integer;
##      end;
##      Mother = record
##          firstname: varying[20] of Char;
##          lastname: varying[20] of Char;
##          num_child: 1..10;
##          children: array[1..10] of Child;
##      end;
```

The following example shows several different declarations of the variable "a_var," illustrating how the same name can be redeclared in nested and parallel scopes, each time referring to a different type:

```
##  procedure Proc_A(a_var: type_1);
##      procedure Proc_B;
##      var
##              a_var: type_2;
##      begin
                {A_var is of type_2}
##      end;
##      function Func_C(a_var: type_3) : Integer;
##      begin
                {Var is of type_3}
##      end;
##  begin
        {A_var is of type_1}
##  end;
```

Special care should be taken when using variables with a **declare cursor** statement. The scope of the variables used in such a statement must also be valid in the scope of the **open** statement for that same cursor. The preprocessor actually generates the code for the **declare** at the point that the **open** is issued, and at that time, evaluates any associated variables.

For example, in the following program fragment, even though the variable "number" is valid to the preprocessor at the point of the **declare cursor** statement, it is not a valid variable name for the Pascal compiler at the point that the **open** is issued.

```
##  procedure Init_Cursor; { Example contains an error }
##  var
##  number: Integer;
##  begin
    { Cursor declaration includes reference to "number" }
##    declare cursor c1 for
##          retrieve (employee.name, employee.age)
##          where employee.num = number
            ...
##  end; { Init_Cursor }

##  procedure Process_Cursor;
##  var
##  ename:  varying[15] of char;
##  eage:   Integer;
##  begin
    { Opening the cursor evaluates invalid "number" }
##  open cursor c1

##  retrieve cursor c1 (ename, eage)
    ...
##  end; { Process_Cursor }
```

Variable Usage

Pascal variables declared to EQUEL can substitute for most elements of EQUEL statements that are not keywords. Of course, the variable and its data type must make sense in the context of the element. To use a Pascal variable in an EQUEL statement, just use its name. To refer to an element, such as a database column, with the same name as a variable, dereference the element by using the EQUEL dereferencing indicator (**#**). As an example of variable usage, the following **retrieve** statement uses the variables "namevar" and "numvar" to receive data, and the variable "idnovar" as an expression in the **where** clause:

```
##  retrieve (namevar = e.name, numvar = e.num)
##          where e.idno = idnovar;
```

You should not use the Pascal type-cast operator (::) in EQUEL statements. The preprocessor ignores it and does not change the type of the variable.

If, in retrieving from Ingres into a program variable, no value is returned for some reason (for example, no rows qualified in a query), the variable will not be modified**.**

Various rules and restrictions apply to the use of Pascal variables in EQUEL statements. The sections below describe the usage syntax of different categories of variables and provide examples of such use.

Simple Variables

A simple scalar-valued variable (integer, floating-point or character string) is referred to by the syntax:

simplename

Syntax Notes:

1. If the variable is used to send data to Ingres, it can be any scalar-valued variable, constant or enumerated literal.
2. If the variable is used to receive data from Ingres, it cannot be a constant or an enumerated literal.
3. Packed or varying arrays of characters (for example, character strings) are referenced as simple variables.

The following program fragment demonstrates a typical message-handling routine that uses two scalar-valued variables, "buffer" and "seconds":

```
##  var
##      buffer : packed array[1..80] of Char;
##      seconds : Integer;
        ...
```

```
##      message buffer
##      sleep seconds
```

A special case of a scalar type is the enumerated type. As mentioned in the section describing declarations, EQUEL treats all enumerated literals and any variables declared with an enumerated type as integers. When used in an EQUEL statement, only the ordinal position of the value in relation to the original enumerated list is relevant. When assigning into an enumerated variable, EQUEL will pass the object by address and assume that the value being assigned into the variable will not raise a runtime error. For example, the following enumerated type declares the states of a table field row, and the variable of that type will always receive one of those values:

```
##  type
##      Table_Field_States =
##          (UNDEFINED, NEWROW, UNCHANGED, CHANGED, DELETED);
##  var
##      tbstate: Table_Field_States;
##      ename: varying[20] of Char;
        ...

        tbstate := undefined;
##      getrow empform employee (ename = name,
##              tbstate = _state)

        case tbstate of
                undefined:
                ...

                deleted:
                ...
        end;
```

Another example retrieves the value TRUE (an enumerated literal of type **boolean**) into a variable when a database qualification is successful:

```
##  var
##          found: Boolean;
##          qual: varying[100] of Char;
            ...

            found := FALSE;
##          retrieve (found = TRUE) WHERE qual

        if not found then
        begin
        ...

        end;
```

Array Variables

An array variable is referred to by the syntax:

*arrayname**[subscript**{,subscript}**]*** {***[subscript**{,subscript}**]***}

Syntax Notes:

1. The variable must be subscripted, because only scalar-valued elements (integers, floating-point and character strings) are legal EQUEL values.
2. When the array is declared, the array bounds specification is not parsed by the EQUEL preprocessor. Consequently, illegal bounds values will be accepted. Also, when an array is referenced, the subscript is not parsed, allowing illegal subscripts to be used. The preprocessor only confirms that an array subscript is used for an array variable. You must make sure that the subscript is legal and that the correct number of indices is used.
3. An array of characters is not a string unless it is **packed** or **varying**.
4. A **packed** or **varying** array of characters is considered a simple variable, not an array variable, in its usage. It therefore cannot be subscripted in order to reference a single character. For example, assuming the following variable declaration and subsequent assignment:

```
##  var
##      abc : packed array[1..3] of Char
    ...
    abc := 'abc';
```

you could not reference

```
abc[1]
```

to access the character "a". To perform such a task, you should declare the variable as a plain (not **packed** or **varying**) array, as, for example:

```
##  var
##      abc : array[1..3] of Char
    ...
    abc := ('a', 'b', 'c');
```

Record Components

The syntax EQUEL uses to refer to a record component is:

record_name{^ | [subscript]}.component{^ | [subscript]}{.component{^ | [subscript]}}

that is, the name of the record, followed by any number of pointer dereference operators or array subscripts, followed by one or more field names (with any number of pointer dereference operators or array subscripts attached).

Syntax Notes:

1. The last record *component* denoted by the above reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays and records, but the last object referenced must be a scalar value. Thus, the following references are all legal:

```
{Assume correct declarations for "employee",}
{ "person"  and other records.}
employee.sal         {Component of a record}
person[3].name       {Component of an element of an array}
rec1.mem1.mem2.age   {Deeply nested component}
```

2. All record components must be fully qualified when referenced. You can shorten the qualification by using the Pascal **with** statement (see below).

3. Any array subscripts or pointer references referred to in the record reference, and not at the very end of the reference, are not checked by the preprocessor. Consequently, both of the following references are accepted, even though one must be wrong, depending on whether "person" is an array:

```
person[1].age
person.age
```

The following example uses the array of records "emprec" to load values into the table field "emptable" in form "empform."

```
##  type
##      EmployeeRec = record
##          ename: packed array[1..20] of Char;
##          eage: [word] -32768 .. 32767;
##          eidno: Integer;
##          ehired: packed array[1..25] of Char;
##          edept: packed array[1..10] of Char;
##          esalary: Real;
##    end;
##  var
##      emprec: array[1..100] of EmployeeRec;
##      i: Integer;

    ...

    for i := 1 to 100 do
    begin
##    loadtable empform emptable
##      (name = emprec[i].ename, age = emprec[i].eage,
##    idno = emprec[i].eidno, hired = emprec[i].ehired,
##    dept = emprec[i].edept, salary = emprec[i].esalary)
    end;
```

The With Statement

You can use the **with** statement to shorten a reference to a record. The syntax of the **with** statement is:

with *record_reference* ***do***
begin
        statements
end [**;**]

where *record_reference* is

record_name{**^** | **[***subscript]*}{*.component*{^ | *[subscript]*}}

that is, the name of a record, followed by any number of pointer dereference operators or array subscripts, followed by zero or more field names (with any number of pointer dereference operators or array subscripts attached).

Following the rules of Pascal,

```
##  with rec_a, rec_b do
##  begin
##      ...
##  end;
```

is exactly equivalent to

```
##  with rec_a do
##  begin
##      with rec_b do
##      begin
##          ...
##      end;
##  end;
```

Syntax Notes:

1. The **with** statement, along with its **begin** and **end** clauses, must be preceded by the EQUEL ## mark in order to be used with EQUEL statements.
2. The *record_reference* must denote a record variable, not a scalar variable.
3. Note that the **with** statement opens the scope of the record so that the member names can stand alone. This creates the possibility that a member name could conflict with the name of an Ingres object. For example, assume that there is an Ingres form called "rname":

```
##  var
##      rec : record
##          rname : packed array[1..12] of char;
##          ri : integer;
##      end;
        ...

##  with rec do
##  begin
##      forminit rname
```

```
##      sleep ri;
##  end;
```

In the **forminit** statement, "rname" refers to "rec.rname," not to the form called "rname," even though outside the scope of the **with** statement it would unambiguously refer to the form. To refer to the form, you must either dereference the name:

```
##  forminit #rname
```

or enclose it in quotes:

```
##  forminit 'rname'
```

The following example uses the array of records "emprec," declared in previous example , to load values into the emptable table field in form "empform."

```
    for i := 1 to 100 do
    begin
##      with emprec[i] do
##      begin
##          loadtable empform emptable
##                (name = ename, age = eage,
##               idno = eidno, hired = ehired,
##               dept = edept, salary = esalary)
##      end;
    end;
```

Pointer Variables

A pointer variable references an object in the same way as in Pascal—the name of the pointer is followed by a caret (^):

pointer_name^

Any further referencing required to fully qualify an object, such as a member of a pointed-to record, follows the usual Pascal syntax.

Syntax Notes:

1. The final object denoted by the pointer reference must be a scalar value (integer, floating-point or character string). There can be any combination of arrays, records or pointer variables, as long as the last object referenced has a scalar value.
2. The pointer reference is also used with file type variables.

In the following example, a pointer to an employee record is used to load a linked list of values into the Employee database table:

```
##  type
##      EmpLink = ^EmployeeRec;
##      EmployeeRec = record
##          ename: packed array [1..20] of Char;
##          eage: Integer;
##          eidno: Integer;
##          enext: EmpLink;
##      end;
##      elist: EmpLink;
        ...

    while (elist <> nil) do
    begin
##          repeat append to employee
##              (name = @elist^.ename, age = @elist^.eage,
##               idno = @elist^.eidno)
            elist := elist^.enext;
    end;
```

Indicator Variables

The syntax for referring to an *indicator* variable is the same as for a simple variable, except that an indicator variable is always associated with a host variable:

host_variable:indicator_variable

Syntax Note:

The indicator variable can be a simple variable, an array element or a record component that yields a 2-byte integer. The type **indicator** has already been declared by the preprocessor. For example:

```
##  var
##      ind_var, ind_arr[5] : Indicator;

    var_1:ind_var
    var_2:ind_arr[2]
```

Data Type Conversion

A Pascal variable declaration must be compatible with the Ingres value it represents. Numeric Ingres values can be set by and retrieved into numeric variables, and Ingres character values can be set by and retrieved into character string variables.

Data type conversion occurs automatically for different numeric types, such as from floating-point Ingres database column values into integer Pascal variables, and for character strings, such as from varying-length Ingres character fields into fixed-length Pascal character string variables.

Ingres does *not* automatically convert between numeric and character types. You must use the Ingres type conversion operators, the Ingres **ascii** function, or a Pascal conversion procedure for this purpose.

The following table shows the default type compatibility for each Ingres data type. Note that some Pascal types do not match exactly and, consequently, may go through some runtime conversion.

Ingres and Pascal Data Type Compatibility

Ingres Type	Pascal Type
c(N), char(N)	**packed array[1..N] of char**
c(N), char(N)	**varying[N] of char**
text(N), varchar(N)	**packed array[1..N] of char**
text(N), varchar(N)	**varying[N] of char**
i1, integer1	**[byte] -128..127**
i2, smallint	**[word] -32768..32767**
i4, integer4	**integer**
f4, float4	**real**
f4, float4	**single**
f8, float8	**double**
date	**packed array[1..25] of char**
money	**double**

Runtime Numeric Type Conversion

The Ingres runtime system provides automatic data type conversion between numeric-type values in the database and forms system and numeric Pascal variables. The standard type conversion rules (according to standard VAX rules) are followed. For example, if you assign a **real** variable to an integer-valued field, the digits after the decimal point of the variable's value are truncated. Runtime errors are generated for overflow on conversion when assigning Ingres numeric values into Pascal variables.

The Ingres **money** type is represented as an 8-byte floating-point value: **double**.

Runtime Character Type Conversion

Automatic conversion occurs between Ingres character string values and Pascal character string variables. There are four string-valued Ingres objects that can interact with character string variables. They are Ingres names, such as form and column names, database columns of type **c**, **char**, **text** or **varchar**, and form fields of type **character**. Several considerations apply when dealing with character string conversions, both to and from Ingres.

The conversion of Pascal character string variables used to represent Ingres names is simple: trailing blanks are truncated from the variables, because the blanks make no sense in that context. For example, the string literals "empform" and "empform" refer to the same form.

The conversion of other Ingres objects is a bit more complicated. First, the storage of character data in Ingres differs according to whether the medium of storage is a database column of type **c** or **character**, a database column of type **text** or **varchar**, or a **character** form field. Ingres pads columns of type **c** or **character** with blanks to their declared length. Conversely, it does not add blanks to the data in columns of type **text** or **varchar**, or in form fields.

Second, the storage of character data in Pascal differs according to whether the character variable is of fixed or varying length. The Pascal convention is to blank-pad fixed-length character strings, but not to pad varying-length character strings. For example, the character string "abc" coming from an Ingres object will be stored in a Pascal **packed array[1..5] of char** variable as the string "abc " followed by two blanks. However, the same string would be stored in a **varying[5] of char** variable as "abc" without any trailing blanks.

When retrieving character data from an Ingres database column or form field into a Pascal variable, you should always ensure that the variable is at least as long as the column or field, in order to avoid truncation of data.

Furthermore, take note of the following conventions:

- Data stored in a database column of type **character** is padded with blanks to the length of the column. The variable receiving such data, be it of fixed or varying length, will contain those blanks. Following Pascal rules, if a fixed-length variable is longer than the database column, the data retrieved into it is further padded with blanks to the length of the variable. In the case of a varying-length variable, no further padding takes place. If the variable is shorter than the database column, truncation of data occurs.
- Data stored in a database column of type **text** or **varchar** is not padded with blanks. If a fixed-length variable is longer than the data in the **text** or **varchar** column, when retrieved, the data is padded with blanks to the length of the variable. In the case of a varying-length variable, no padding takes place. If the variable is shorter than the database column, truncation of data occurs.

- Data stored in a **character** form field contains no trailing blanks. If a fixed-length variable is longer than the data in the field, when retrieved, the data is padded with blanks to the length of the variable. In the case of a varying-length variable, no padding takes place. If the variable is shorter than the field, truncation of data occurs.

When inserting character data into an Ingres database column or form field from a Pascal variable, note the following conventions:

- When data is inserted from a Pascal variable into a database column of type **c** or **character** and the column is longer than the variable, the column is padded with blanks. If the column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a Pascal variable into a database column of type **text** or **varchar** and the column is longer than the variable, no padding of the column takes place. Furthermore, by default, all trailing blanks in the data are truncated before the data is inserted into the **text** column. For example, when a string "abc" stored in a Pascal **packed array[1..5] of char** variable as "abc " is inserted into the **text** or **varchar** column, the two trailing blanks are removed and only the string "abc" is stored in the database column. To retain such trailing blanks, you can use the EQUEL **notrim** function. It has the following syntax:

  notrim(*stringvar*)

  where *stringvar* is a character string variable. An example demonstrating this feature follows later. When used with **repeat** queries, the **notrim** syntax is:

  @notrim(*stringvar*)

  If the **text** or **varchar** column is shorter than the variable, the data is truncated to the length of the column.
- When data is inserted from a Pascal variable into a **character** form field and the field is longer than the variable, no padding of the field takes place. In addition, all trailing blanks in the data are truncated before the data is inserted into the field. If the field is shorter than the data (even after all trailing blanks have been truncated), the data is truncated to the length of the field.

  When comparing character data in an Ingres database column with character data in a Pascal variable, note the following convention:
- When comparing data in **c**, **character**, or **varchar** database columns with data in a character variable, all trailing blanks are ignored. Trailing blanks are significant in **text**. Initial or embedded blanks are significant in **character**, **text**, and **varchar**; they are ignored in **c**.

As described above, the conversion of character string data between Ingres objects and Pascal variables often involves the trimming or padding of trailing blanks, with resultant change to the data. If trailing blanks have significance in your application, give careful consideration to the effect of any data conversion. For information on the significance of blanks when comparing with various Ingres character types, see the *QUEL Reference Guide*.

The Ingres **date** data type is represented as a 25-byte character string.

The program fragment in the following example demonstrates the **notrim** function and the truncation rules explained above.

```
{
| Assume that a table called "textchar" has been created
| with the following CREATE statement:
|
|   CREATE textchar
|            (row = i4,
|             data = text(10)) -- Note the text data type
}

##  var
##      row: Integer;
##      p_data: packed array[1..7] of Char;
##      v_data: varying[7] of Char;
        ...

        p_data := 'abc  '; {Holds "abc "}
        v_data := 'abc'; {Holds "abc"}

        {The following APPEND adds the string "abc" (blanks truncated)}
##  append to textchar (#row = 1, #data = p_data)

{The following APPEND adds the string "abc" (never had blanks)}
##  append to textchar (#row = 2, #data = v_data)

{
| This statement adds the string "abc ", with 4 trailing
| blanks left intact by using the NOTRIM function.
}
##   append to textchar (#row = 3, #data = notrim(p_data))

{
| This RETRIEVE retrieves rows #1 and #2, because
| trailing blanks were suppressed when these rows were
| appended.
}

##  retrieve (row = textchar.#row)
##           where length (textchar.#data) = 3
##  begin
            writeln( 'row found = ', row );
##  end;

{
| This RETRIEVE retrieves row #3, because the NOTRIM
| function left trailing blanks in the "data" variable
| in the last APPEND statement.
}
```

```
##  retrieve (row = textchar.#row)
##           where length (textchar.#data) = 7
##  begin
            writeln( 'row found = ', row );
##  end;
```

Dynamically Built Param Statements

The **param** feature dynamically builds EQUEL statements. EQUEL/Pascal does not currently support **param** versions of statements. **Param** statements are supported in EQUEL/C and EQUEL/Fortran.

Runtime Error Processing

This section describes a user-defined EQUEL error handler.

Programming for Error Message Output

By default, all Ingres and forms system errors are returned to the EQUEL program, and default error messages are printed on the standard output device. As discussed in the *QUEL Reference Guide*, you can also detect the occurrences of errors by means of the program using the **inquire_ingres** and **inquire_frs** statements. Use the latter for checking errors after forms statements. Use **inquire_ingres** for all other EQUEL statements.

This section discusses an additional technique that enables your program not only to detect the occurrences of errors but also to suppress the printing of default Ingres error messages if you choose. The **inquire** statements detect errors but do not suppress the default messages.

This alternate technique entails creating an error-handling function in your program and passing its address to the Ingres runtime routines. Then Ingres will automatically invoke your error handler whenever an Ingres or a forms-system error occurs. Your program error handler must be declared as follows:

```
[global] function funcname (ingerr:Integer):Integer;
    begin
...
    end;
```

This function must be passed to the EQUEL routine **IIseterr( )** for runtime bookkeeping using the statement:

```
IIseterr(%immed funcname);
```

This forces all runtime Ingres errors through your function, passing the Ingres error number as an argument. If you choose to handle the error locally and suppress Ingres error message printing, the function should return 0; otherwise the function should return the Ingres error number received.

Avoid issuing any EQUEL statements in a user-written error handler defined to **IIseterr**, except for informative messages, such as **message**, **prompt**, **sleep**, and **clear screen**, and messages that close down an application, such as **endforms** and **exit.**

The following example demonstrates a typical use of an error function to warn users of access to protected tables.

```
## program ErrorHandling(input, output);

    ...

    [global] function ErrorProc(ingerr: Integer) : Integer;
        const
        TBLPROT = 5003;
    begin
    if (ingerr = TBLPROT) then begin
        writeln('You are not authorized for this operation');
        ErrorProc := 0;   { Suppress Ingres message }
    end else begin
        ErrorProc := ingerr; { Ingres will print message }
        end;
    end;                { ErrorProc }

## declare

## begin

##      Ingres dbname

        ...

        IIseterr(%immed ErrorProc);

        ...

##  end.                 { ErrorHandling}
```

Precompiling, Compiling, and Linking an EQUEL Program

This section describes the EQUEL preprocessor for Pascal and the steps required to precompile, compile, and link an EQUEL program.

Generating an Executable Program

Once you have written your EQUEL program, it must be preprocessed to convert the EQUEL statements into Pascal code. This section describes the use of the EQUEL preprocessor. Additionally, it describes how to compile and link the resulting code to obtain an executable file.

The EQUEL Preprocessor Command

The Pascal preprocessor is invoked by the following command line:

eqp {*flags*} {*filename*}

where *flags* are

Flag	Description
-d	Adds debugging information to the runtime database error messages generated by EQUEL. The source file name, line number, and the erroneous statement itself are printed with the error message.
-f[*filename*]	Writes preprocessor output to the named file. If the **-f** flag is specified without a *filename*, the output is sent to standard output, one screen at a time. If the **-f** flag is omitted, output is given the basename of the input file, suffixed ".pas".
-l	Writes preprocessor error messages to the preprocessor's listing file, as well as to the terminal. The listing file includes preprocessor error messages and your source text in a file named *filename**.lis***, where *filename* is the name of the input file.
-lo	Like **-l**, but the generated Pascal code also appears in the listing file.
-n. ***ext***	Specifies the extension used for filenames in **## include** and ## include inline statements in the source code. If **-n** is omitted, include filenames in the source code must be given the extension ".qp".
-o	Directs the preprocessor not to generate output files for include files. This flag does not affect the translated **include** statements in the main program. The preprocessor will generate a default extension for the translated include files statements unless you use the **-o**.*ext* flag.
-o. *ext*	Specifies the extension given by the preprocessor to both the translated **include** statements in the main program and the generated output files. If this flag is not provided, the default extension is ".pas". If you use this flag in combination with the **-o** flag, then the preprocessor generates the specified extension for the translated include statements, but does not generate new output files for the include statements.

Flag	Description
-s	Reads input from standard input and generates Pascal code to standard output. This is useful for testing statements you are not familiar with. If the **-l** option is specified with this flag, the listing file is called "stdin.lis". Type **Ctrl Z** to terminate the interactive session**.**
-w	Prints warning messages.
-?	Shows the available command line options for **eqp**.

The EQUEL/Pascal preprocessor assumes that input files are named with the extension ".qp". This default can be overridden by specifying the file extension of the input file(s) on the command line. The output of the preprocessor is a file of generated Pascal statements with the same name and the extension ".pas".

If you enter the command without specifying any flags or a filename, Ingres displays a list of flags available for the command.

The following table presents the options available with **eqp**.

Eqp Command Examples

Command	Comment
eqp file1	Preprocesses "file1.qp" to "file1.pas"
eqp -l file2.xp	Preprocesses "file2.xp" to "file2.pas" and creates listing "file2.lis"
eqp -s	Accepts input from standard input and writes generated code to standard output
eqp -ffile4.out file4	Preprocesses "file4.qp" to "file4.out"
eqp	Displays a list of flags available for this command.

The Pascal Compiler

As mentioned above, the preprocessor generates Pascal code. You should use the VMS **Pascal** command to compile this code. Most of the **Pascal** command line options can be used. You must not use the **g_floating** qualifier if real variables in the file are interacting with Ingres floating-point objects. You should also not use the **old_version** qualifier, because the preprocessor generates code for Version 3. Note, too, that many of the statements that the EQUEL/Pascal preprocessor generates are nonstandard extensions provided by VAX/VMS. Consequently, you should not use the **standard** qualifier.

The following example preprocesses and compiles the file "test1." Note that both the EQUEL/Pascal preprocessor and the Pascal compiler assume the default extensions:

```
$ eqp test1
$ Pascal/list test1
```

Installing the EQUEL/Pascal Environment File

As explained in The Inherit Attribute, EQUEL/Pascal programs can inherit the EQUEL/Pascal declarations from an environment file, as an alternative to the **declare** statement. If the program specifies this alternative, the preprocessor will generate an "Inherit" attribute referencing this environment file. The file is named "eqenv.pen" and is located in the Ingres files directory, which, by default, is "ii_system:[ingres.files]".

Before using the environment file, you should ensure that your System Administrator has installed it, using the following sequence of operating system commands:

```
$ set def ii_system:[ingres.files]
$ eqp eqenv
$ Pascal eqenv
$ delete eqenv.pas;*, eqenv.obj;*
```

Note: Check the Readme file for any operating system specific information on compiling and linking EQUEL/Pascal programs.

Linking an EQUEL Program

EQUEL programs require procedures from several VMS shared libraries in order to run properly. Once you have preprocessed and compiled an EQUEL program, you can link it. Assuming the object file for your program is called "dbentry," use the following link command:

```
$ link dbentry.obj,-
  ii_system:[ingres.files]equel.opt/opt
```

It is recommended that you do not explicitly link in the libraries referenced in the EQUEL.OPT file. The members of these libraries change with different releases of Ingres. Consequently, you may be required to change your link command files in order to link your EQUEL programs.

Assembling and Linking Pre-Compiled Forms

The technique of declaring a pre-compiled form to the FRS is discussed in the *QUEL Reference Guide*. To use such a form in your program, you must also follow the steps described here.

In VIFRED, you can select a menu item to compile a form. When you do this, VIFRED creates a file in your directory describing the form in the VAX-11 MACRO language. VIFRED lets you select the name for the file. Once you have created the MACRO file this way, you can assemble it into linkable object code with the VMS command

macro *filename*

The output of this command is a file with the extension ".obj". You then link this object file with your program (in this case named "formentry" by listing it in the link command, as in the following example:

```
$ link formentry,-
  empform.obj,-
  ii_system:[ingres.files]equel.opt/opt
```

Linking an EQUEL Program without Shared Libraries

While the use of shared libraries in linking EQUEL programs is recommended for optimal performance and ease-of-maintenance, non-shared versions of the libraries have been included in case you require them. Non-shared libraries required by EQUEL are listed in the equel.noshare options file. The options file must be included in your link command *after* all user modules. Libraries must be specified in the order given in the options file.

The following example demonstrates the link command of an EQUEL program called "dbentry" that has been preprocessed and compiled:

```
$ link dbentry,-
  ii_system:[ingres.files]equel.noshare/opt
```

Include File Processing

The EQUEL **include** statement provides a means to include external files in your program's source code. Its syntax is:

include *filename*

Filename is a quoted string constant specifying a file name, or a logical name that points to the file name. You must use the default extension ".qp" on names of **include** files, unless you override this requirement by specifying a different extension with the **-n** flag of the **eqp** command.

This statement is normally used to include variable declarations, although it is not restricted to such use. For more details on the **include** statement, see the *QUEL Reference Guide*.

The included file is preprocessed and an output file with the same name but with the default output extension ".pas" is generated. You can override this default output extension with the **-o**.*ext* flag on the command line. The reference in the original source file to the included file is translated in the output file to the specified include output file. If the **-o** flag is used with no extension, no output file is generated for the include file. This is useful for program libraries that are using VMS MMS dependencies.

If you use both the **-o**.*ext* and the **-o** flags, then the preprocessor will generate the specified extension for the translated **include** statements in the program but will not generate new output files for the statements.

For example, assume that no overriding output extension was explicitly given on the command line. The EQUEL statement:

```
## include 'employee.qp'
```

is preprocessed to the Pascal statement:

```
% include 'employee.pas'
```

and the file "employee.qp" is translated into the Pascal file "employee.pas".

As another example, assume that a source file called "inputfile" contains the following **include** statement:

```
## include 'mydecls'
```

The name "mydecls" is defined as a system logical name pointing to the file "dra1:[headers]myvars.qp" by means of the following command at the DCL level:

```
$ define mydecls dra1:[headers]myvars.qp
```

Assume now that "inputfile" is preprocessed with the command:

```
$ eqp -o.h inputfile
```

The command line specifies ".h" as the output file extension for include files. As the file is preprocessed, the **include** statement shown earlier is translated into the Pascal statement:

```
% include 'dra1:[headers]myvars.h'
```

and the Pascal file "dra1:[headers]myvars.h" is generated as output for the original include file, "dra1:[headers]myvars.qp".

For including source code using the **include** inline statement, see the *QUEL Reference Guide*.

You can also specify include files with a relative path. For example, if you preprocess the file "dra1:[mysource]myfile.qp," the EQUEL statement:

```
## include '[-.headers]myvars.qp'
```

is preprocessed to the Pascal statement:

```
%include '[-.headers]myvars.pas'
```

and the Pascal file "dra1:[headers]myvars.pas" is generated as output for the original include file, "dra1:[headers]myvars.qp."

Including Source Code with Labels

Some EQUEL statements generate labels in the output code. If you include a file containing such statements, you must be careful to include the file only once in a given Pascal scope. Otherwise, you may find that the compiler later issues Pascal warning or error messages to the effect that the generated labels are multiply defined in that scope.

The statements that generate labels are the **retrieve** statement and all the EQUEL/FORMS block-type statements, such as **display** and **unloadtable**.

Coding Requirements for Writing EQUEL Programs

The following sections describe coding requirements for writing EQUEL programs.

Comments Embedded in Pascal Output

Each EQUEL statement generates one comment and a few lines of Pascal code. You may find that the preprocessor translates 50 lines of EQUEL into 200 lines of Pascal. This may result in confusion about line numbers when you are debugging the original source code. To facilitate debugging, each group of Pascal statements associated with a particular statement is preceded by a comment corresponding to the original EQUEL source. (Note that only *executable* EQUEL statements are preceded by a comment.) Each comment is one line long and informs the reader of the file name, line number, and type of statement in the original source file.

One consequence of the generated comment is that you cannot comment out embedded statements by putting the opening comment delimiter on an earlier line. You have to put the opening comment delimiter on the same line, before the **##** delimiter, to cause the preprocessor to treat the complete statement as a Pascal comment.

The Pascal Semicolon and EQUEL Statements

With one exception, EQUEL statements embedded in Pascal host code do not require a terminating semicolon. Pascal declarative statements *must* be separated by a semicolon, as required in the Pascal language.

The exception occurs when an EQUEL statement that allows but does not include the optional **with** clause is followed immediately by a Pascal **with** statement. When this occurs, the EQUEL statement must be terminated with a semicolon. For example:

```
##  {Assume "emprec" has been declared as a
##   record variable}
##  create employee (name=c30, age=i4);
##  {Note the semicolon here}
##  with emprec do
##  begin
    ...
##  end;
```

If the EQUEL statement with the optional **with** clause is followed by another EQUEL statement or by Pascal host code, then the semicolon is optional.

Pascal Blocks Generated by EQUEL

As mentioned above, the preprocessor may produce several Pascal statements for a single EQUEL statement. However, all the Pascal statements that the preprocessor generates for an EQUEL statement are surrounded by a **begin-end** block. Thus, the statement:

```
    if error then
##  deleterow form table 1
```

will produce legal Pascal code, even though the **deleterow** statement generates more than one Pascal statement.

Note that multiple EQUEL statements will cause the preprocessor to generate multiple **begin-end** blocks. Therefore, when placing multiple EQUEL statements in a Pascal **if** statement, you must surround the whole group of statements with a **begin-end** block, just as you would for multiple Pascal statements in an **if** statement. For example:

```
    if error then
    begin
##      message 'Deleting because of error'
##      sleep 2
##      deleterow form table 1
    end;
```

A semicolon always terminates the **begin-end** block that the preprocessor generates for an EQUEL statement. Therefore, because Pascal does not permit semicolons before the **else** clause of an **if** statement, you must surround any single EQUEL statement that precedes an **else** clause with a **begin-end** block. For example, the following **if** statement will cause a Pascal error:

```
    if error then
##      message 'Error occurred'
        {Preprocessor adds a semicolon here}
    else
##      message 'No error occurred'
```

By delimiting the **then** clause with **begin-end**, you eliminate the error:

```
    if error then
    begin
##      message 'Error occurred'
        {Preprocessor still adds semicolon here...}
        end
        {...but that's okay because there's no semicolon here}
    else
##      message 'No error occurred'
```

An EQUEL Statement that Does Not Generate Code

The **declare cursor** statement does not generate any Pascal code. This statement should not be coded as the only statement in Pascal constructs that does not allow *null* statements. For example, coding a **declare cursor** statement as the only statement in a Pascal **if** statement not bounded by **begin** and **end** would cause compiler errors:

```
    if (using_database)
##  declare cursor empcsr for retrieve (employee.ename)
    else
        writeln('You have not accessed the database.');
```

The code generated by the preprocessor would be:

```
    if (using_database)
    else
        writeln('You have not accessed the database.');
```

which is an illegal use of the Pascal **else** clause.

EQUEL/Pascal Preprocessor Errors

To correct most errors, you may wish to run the EQUEL preprocessor with the listing (**-l**) option on. The listing will be sufficient for locating the source and reason for the error.

For preprocessor error messages specific to the Pascal language, see the next section.

Preprocessor Error Messages

The following is a list of error messages specific to the Pascal language.

E_E20001 "Pascal attribute conflict in declaration of size for '%0c'."

Explanation: The program has specified conflicting size attributes for this object. For example, the following declaration is erroneous because of the attempt to extend the attribute size of the type 'smaller':

```
type
          smaller = [byte] 1..100;
var
          bigger : [word] smaller;
```

E_E20002 "Pascal subrange conflict. Upper and lower bounds are not the same type or they are not an ordinal type."

Explanation: Both bounds of a subrange declaration must be of the same ordinal type (single character or integer). If the subrange bounds types are different or if they are not ordinal types, the preprocessor will use the type of the second bound and accept the usage of variables declared with this subrange type. This will cause an error in later Pascal compilation.

E_E20003 "Mismatching statement at end of Pascal subprogram. Check balanced subprogram headers and END pairs."

Explanation: You may have an **end** statement that is not balanced by a subprogram header (for example, PROGRAM, PROCEDURE, FUNCTION, or MODULE). These subprogram delimiters provide scoping for Pascal variables and labels generated by the preprocessor. If you had any syntax errors on the subprogram header statement, then correct those errors and preprocess the file again.

E_E20004 "No ## DECLARE before first EQUEL statement `%0c'."

Explanation: You must issue the **## declare** statement before the first embedded statement. The preprocessor generates code that references procedures and functions declared in a file included by the **## declare** statement. Without issuing the **## declare** statement, the Pascal compiler will not accept those references.

E_E20005 "Pascal character array '%0c' must be PACKED or VARYING."

Explanation: A string referenced in an embedded statement must be either a **packed array of char**, a **varying of char**, or a single **char**. You have used a non-packed **array of char** as an embedded string variable. Convert the variable declaration to either **packed** or **varying**, or subscript the array to reference only one element.

E_E20006 "Extraneous semicolon in Pascal declaration ignored."

Explanation: Only one semicolon is allowed between components of a record declaration. The preprocessor ignores the extra semicolons. You should delete the extra semicolon in your source code.

E_E20007 "Pascal dimension of '%0c' is %1c, but subscripted %2c times."

Explanation: You have not referenced the specified variable with the same number of subscripts as the number of dimensions with which the variable was declared. This error indicates that you have failed to subscript an array, or you have subscripted a non-array. The preprocessor does not parse declaration dimensions or subscript expressions.

E_E20008 "Incorrect indirection of Pascal variable '%0c'. Variable is declared with indirection of %1c, but dereferenced (^) %2c time(s)."

Explanation: This error occurs when the address or value of a variable is incorrectly expressed because of faulty indirection. For example, the name of an integer pointer has been given instead of the variable that the pointer was pointing at. Either redeclare the variable with the intended indirection (and check any implicit indirection in the type), or change its use in the current statement.

E_E20009 "Pascal Pass 2 failure on INCLUDE file. The maximum INCLUDE nesting exceeded %0c."

Explanation: The Pascal preprocessor must take a second pass in order to declare implicitly generated labels. If the source file referenced embedded INCLUDE files, then the second pass needs to generate labels into those files. Consequently there is a maximum nesting limit of INCLUDE files. Try reorganizing your files to create a flatter source file structure.

E_2000A "No ## PROCEDURE for current scope but labels have been generated."

Explanation: The Pascal preprocessor must take a second pass in order to declare implicitly generated labels. If labels were implicitly generated then the preprocessor needs to know where to declare them on the second pass. That is why one must precede subprogram headers (PROGRAM, PROCEDURE, FUNCTION and MODULE) with ##, or use the LABEL statement. If you did not declare your subprogram header to the preprocessor, the generated labels will be marked as undeclared by the Pascal compiler.

E_E2000B "Pascal Pass 2 open file failure. Cannot pass information from file '%0c' to '%1c'."

Explanation: The Pascal preprocessor must take a second pass in order to declare implicitly generated labels. Because there is a temporary file involved, and this file has a fixed name, you should avoid running the preprocessor more than once in the same directory. This error may also occur if the intermediate file disappeared, the system protections of the current directory are too restrictive or have changed, or if the original input file was moved between the first and second pass of the preprocessor.

E_E2000C "Pascal Pass 2 file inconsistency. Mismatching number of label markers in '%0c'."

Explanation: The Pascal preprocessor must take a second pass in order to declare implicitly generated labels. There was a difference between the number of label declaration sections the preprocessor expected to generate and the number of markers found in the intermediate file. This may be caused by an embedded **include** statement that requires its own scope for label generation. If there were nested **include** statements whose files required labels, try to flatten them out into larger source files.

E_E2000D "Missing Pascal keyword '%0c' in declaration."

Explanation: You did not use the specified keyword, or you did not make the word known to the preprocessor. If there are no other errors the preprocessor will generate correct Pascal code.

E_E2000E "Illegal nesting of Pascal compilation units."

Explanation: You cannot nest modules and programs in themselves or each other. Make sure you have placed the ## mark before the **end** statement for programs and modules.

E_E2000F "Can not use indirection (^) on an undeclared Pascal variable '%0c'."

Explanation: You have used pointer indirection on a name that was not declared as a Pascal variable to the preprocessor. If this really is a variable you should make its declaration known to the preprocessor.

E_E20010 "Can not subscript ([]) an undeclared Pascal variable '%0c'."

Explanation: You have used array subscription on a name that was not declared as a Pascal variable to the preprocessor. If this really is a variable you should make its declaration known to the preprocessor.

E_E20011 "Can not subscript VARYING Pascal variable '%0c'."

Explanation: Elements of a varying-length character string array cannot be passed to the runtime system. If you need to pass a single element then declare the array as a plain array (not PACKED nor VARYING).

E_E20012 "Scalar Pascal type required for conformant schema bounds type."

Explanation: Pascal requires that bounds expressions of conformant arrays be of a scalar type. You must choose a scalar type, such as a single character or an integer.

E_E20013 "Pascal object '%0c' is not a variable."

Explanation: You have used the specified name as an embedded variable, but you have not declared it to the preprocessor. This may also be a scope problem. Make sure you have typed the name correctly, declared the variable to the preprocessor and have used it in its scope.

E_E20014 "Too many comma separated names in declaration. Maximum number of names is %0c."

Explanation: The declaration of a comma-separated list of names in a declaration is too long. For example:

```
var
    a, b, ..... N : Integer;
```

Try breaking up the declaration into groups.

E_E20015 "EQUEL/Pascal does not support PARAM target lists."

Explanation: If you need to use PARAM target lists, then you should write this subprogram in another host language (such as C or Fortran) and link that module with your Pascal program.

E_E20016 "Reissue of ## DECLARE statement. Second time is ignored."

Explanation: The ## DECLARE statement should occur only once per module. Placing the statement after an EQUEL statement will also cause this error.

E_E20017 "Missing semicolon (;) at end of Pascal LABEL declaration list."

Explanation: Earlier versions of EQUEL/Pascal did not require the use of a semicolon after the **label** statement. The preprocessor now requires the terminating semicolon if you include a list of your own labels with the **label** statement. If you do not include the semicolon, the preprocessor will generate correct code, but you should still correct the error.

E_E20018 "Last Pascal record member referenced in '%0c' is unknown."

Explanation: The last record member referenced is not a member of the current record. Make sure you have spelled the member name correctly, and that it is a member of the specified record.

E_E20019 "Unclosed Pascal block. There are %0c unbalanced subprogram headers."

Explanation: The end of the file was reached with some program blocks left open. Make sure you have an **end** statement for each subprogram header or embedded LABEL statement.

E_E2001A "Pascal %0c '%1c' is not yet defined. An INTEGER is assumed."

Explanation: The specified TYPE or CONST name has not yet been declared. Make sure that all types and constants are defined before use. Forward type declarations (such as pointers to undefined types) are an exception.

E_E2001B "Underflow of comma separated name list in declaration."

Explanation: The stack used to store comma-separated names in declarations has been corrupted. Try rearranging the list of names in the declaration.

E_E2001C "Pascal variable '%0c' is of unsupported type SET or QUADRUPLE."

Explanation: You may declare variables of type **set** and **quadruple**, but you may not use them in embedded statements. The declarations are only allowed so that you can declare records with components of those types. If those variables need to interact with INGRES, then declare the **set** variable as an **array of boolean**, and the **quadruple** variable as a **double**.

E_E2001D "Adding an unknown name '%0c' in Pascal WITH statement."

Explanation: The specified name is not known to the preprocessor when used with an embedded **with** statement. Check its spelling and make sure it was declared to the preprocessor in the correct scope.

E_E2001E "Overflow of Pascal WITH stack on variable '%0c'. Maximum depth is %1c."

Explanation: You have nested embedded **with** blocks too deeply. Flatten your record declarations, or use partially qualified names in place of the deepest **with** statement.

E_E2001F "A Pascal WITH block is still open."

Explanation: Every **with** block must be closed by an **end** statement. This error indicates that the end of a routine has been encountered before a **with** block inside the routine has been ended.

E_E20020 "Pascal WITH variable '%0c' must be of type RECORD."

Explanation: A **with** statement specified a variable that was not a record. Check the name and verify that the scoping rules ensure that this use of the specified name refers to a record variable.

E_E20021 "Underflow of Pascal WITH stack."

Explanation: The stack used to manage a **with** record has been corrupted. Try rearranging the nesting of **with** statements, or partially qualify some of the more deeply nested record components.

E_E20022 "Pascal variable '%0c' is a record, not a scalar value."

Explanation: The named variable refers to a record. It was used where a variable must be used to retrieve data from INGRES. This error may also cause a syntax error on any subsequent record components that are referenced.

Sample Applications

This section contains sample applications.

The Department-Employee Master/Detail Application

This application uses two database tables joined on a specific column. This typical example of a department and its employees demonstrates how to process two tables as a master and a detail.

The program scans through all the departments in a database table, in order to reduce expenses. Department information is stored in program variables. Based on certain criteria, the program updates department and employee records. The conditions for updating the data are the following:

Departments:

- If a department has made less than $50,000 in sales, the department is dissolved.

Employees:

- If an employee was hired since the start of 1985, the employee is terminated.
- If the employee's yearly salary is more than the minimum company wage of $14,000 and the employee is not nearing retirement (over 58 years of age), the employee takes a 5% pay cut.
- If the employee's department is dissolved and the employee is not terminated, the employee is moved into a state of limbo (the "toberesolved" database table, described below) to be resolved by a supervisor.

This program uses two cursors in a master/detail fashion. The first cursor is for the Department table, and the second is for the Employee table. The **create** statements used to create the tables are shown below. The cursors retrieve all the information in their respective tables, some of which is updated. The cursor for the Employee table also retrieves an integer date interval whose value is positive if the employee was hired after January 1, 1985.

Each row that is scanned, both from the Department table and the Employee table, is recorded into the system output file. This file serves as a log of the session and as a simplified report of the updates that were made.

Each section of code is commented for the purpose of the application and also to clarify some of the uses of the EQUEL statements. The program illustrates table creation, multi-query transactions, all cursor statements and direct updates. For purposes of brevity, error handling on data manipulation statements is simply to close down the application.

The following two **create** statements describe the Employee and Department database tables:

```
##      create dept
##               (name     = c12,   { Department name }
##               totsales  = money, { Total sales }
##               employees = i2)    { Number of employees }

##      create employee
##               (name     = c20,   { Employee name }
##                age      = i1,    { Employee age }
##                idno     = i4     { Unique employee id }
##                hired    = date,  { Date of hire }
##                dept     = c10,   { Employee department }
##                salary   = money) { Yearly salary }

##      program Departments( input, output );

##      type
##          String12 = varying[12] of Char;
##          String20 = varying[20] of Char;
##          String25 = varying[25] of Char;
##          String132 = varying[132] of Char;
##          Short_Short_Integer = [Byte] -128 .. 127;
##          Short_Integer = [Word] -32768 .. 32767;
##          Long_Float = Double;
##      label
##          Exit_Program;
##      DECLARE

        {
        | Function: Close_Down
        | Purpose:  If an error occurs during the execution of an
        |            EQUEL statement this error handler is called.
        |            Errors are printed and the current database session is
        |            terminated. Any open transactions are implicitly closed.
        | Parameters:
        |               ingerr - Integer containing Ingres error number.
         }
```

```
##      [global] function Close_Down(ingerr: Integer): Integer;
##      var
##              err_text: varying [200] of char;
##      begin {Close_Down}
##              inquire_ingres (err_text = errortext)
##              exit
                Writeln('Closing down because of database error:');
                Writeln(err_text);
                Close_Down := ingerr;
                goto Exit_Program;
##      end; {Close_Down}

        {
        | Procedure:    Process_Expenses -- MAIN
        | Purpose:      Main body of the application. Initialize
        |               the database,
        |               process each department, and terminate the session.
        | Parameters:
        |               None
        }

##       procedure Process_Expenses;

                {
                | Function:     Init_Db
                | Purpose:      Initialize the database. Connect to the
                |               database, and abort on error. Before
                |               processing departments and employees create
                |               the table for employees who lose their
                |               department, "toberesolved". Initiate the
                |               multi-statement transaction.
                | Parameters:
                |               None
                | Returns:
                |               FALSE - Failed to start application.
                |               TRUE - Succeeded in starting application.
                }

##              Function Init_Db : Boolean;
##              var
##                      create_err: Integer;
##              begin {Init_Db}

##                      Ingres personnel

                        {Create the table}
                        Writeln('Creating "To_Be_Resolved" table.');
##                      create toberesolved
##                              (name = c20,
##                               age = smallint,
##                               idno = integer,
##                               hired = date,
##                               dept = c10,
##                               salary = money)
##              inquire_ingres (create_err = ERRORNO)
```

```
                    if (create_err > 0) then begin
                    Writeln('Fatal error creating application table.');
                             Init_Db := FALSE;
                    end else begin
                            {
                            | Inform Ingres runtime system about error handler
                            | All errors from here on close down
                            | the application.
                            }
                            IIseterr(%immed Close_Down);
##                          begin transaction
                            Init_Db := TRUE;
                             end; {If create error}

##                  end; {Init_Db}

                    {
                    | Procedure: End_Db
                    | Purpose:   Commit the multi-statement transaction and
                    |            end access to the database after successful
                    |            completion of the application.
                    | Parameters:
                    |            None
                    }

##                  Procedure End_Db;
##                  begin {End_Db}
##                          end transaction
##                          exit
##                  end; {End_Db}
                    {
                    | Procedure:    Process_Employees
                    | Purpose:      Scan through all the employees for a
                    |               particular department. Based on given
                    |               conditions the employee may be terminated,
                    |               or take a salary reduction.
                    |               1. If an employee was hired since 1985 then
                    |                  the employee is terminated.
                    |               2. If the employees yearly salary is more
                    |                  than the minimum company wage of $14,000
                    |                  and the employee is not close
                    |                  to retirement
                    |                  (over 58 years of age), then the employee
                    |                  takes a 5% salary reduction .
                    |               3. If the employee's department is dissolved
                    |                  and the employee is not terminated, then
                    |                  the employee is moved into the
                    |                  "toberesolved" table.
                    | Parameters:
                    |               dept_name - Name of current department.
                    |               deleted_dept - Is current department being
                    |               dissolved?
                    |               emps_term - Set locally to record how many
                    |               employees were terminated
                    |               for the current department.
                    }
```

```
##              procedure Process_Employees (dept_name: String12;
##                                           deleted_dept: Boolean;
##                                           var emps_term: Integer);
##              const
##                       SALARY_REDUC = 0.95;
##                       MIN_EMP_SALARY = 14000.00;
##                       NEARLY_RETIRED = 58;
##              type
                        {Emp_Rec corresponds to the "employee" table}
##                      Emp_Rec = record
##                              name:   String20;
##                              age:    Short_Short_Integer;
##                              hired:  String25;
##                              idno:   Integer;
##                              salary: Real;
##                              hired_since_85: Integer;
##                      end; {record}
##              var
##                      emp:            Emp_Rec;
                        title:          String12; {Formatting values}
                        description:    String25;
##                      no_rows:        Integer;
##              begin {Process_Employees}

                        {
                        | Note the use of the Ingres function to find out
                        | who was hired since 1985.
                        }

##                      range of e IS employee

##                      declare cursor empcsr FOR
##                       retrieve (e.name, e.age, e.idno, e.hired, e.salary,
##                                 res = int4(interval('days',
##                                 e.hired-date('01-jan-1985'))))
##                       where e.dept = dept_name
##                       for direct update of (name, salary)

                        no_rows := 0;
                        emps_term := 0; {Record how many}

##                      open cursor empcsr

                        while (no_rows = 0) do begin
##                       retrieve cursor empcsr (emp.name, emp.age, emp.idno,
##                                               emp.hired, emp.salary,
##                                               emp.hired_since_85)
##                       inquire_equel (no_rows = endquery)
                         if (no_rows = 0) then begin

                         {Terminate new employees}
                          if (emp.hired_since_85 > 0) then begin

##                              delete cursor empcsr
                                title := 'Terminated: ';
                                description := 'Reason: Hired since 1985.';
                                emps_term := emps_term + 1;

                        {Else reduce salary if large and not nearly retired}
                        end else if (emp.salary > MIN_EMP_SALARY) then begin
```

```
                        if (emp.age < nearly_retired) then begin
##                       replace cursor empcsr
##                              (salary = salary * salary_reduc)
                         title := 'Reduction: ';
                         description := 'Reason: Salary.';
                         end else begin
                                  {Do not reduce salary - nearly retired}
                                 title := 'No Changes: ';
                                 description := 'Reason: Retiring.';
                         end; {If retiring}

                                {Else leave employee alone - low salary}
                                end else begin

                                 title := 'No Changes: ';
                                  description := 'Reason: Salary.';
                                end;

                                {Was employee's department dissolved?}
                                if (deleted_dept) then begin
##                                      append to toberesolved (e.all)
##                                         where e.idno = emp.idno
##                                      delete cursor empcsr
                                end;

                                {Log the employee's information}
                                 Write(' ', title, ' ', emp.idno:6, ', ');
                                 Write(emp.name, ', ', emp.age:3, ', ');
                                 Writeln(emp.salary:8:2, ' ; ', description);

                          end; {If a row was retrieved}

                        end; {Continue with cursor loop}

##                      close cursor empcsr

##              end; {Process_Employees}

                {
                | Procedure: Process_Depts
                | Purpose: Scan through all the departments, processing
                | each one. If the department has made less
                | than $50,000 in sales, then the department
                | is dissolved.
                | For each department process all the
                | employees (they may even be moved to another
                | database table).
                | If an employee was terminated, then update
                | the department's employee counter.
                | Parameters:
                | None
                }
```

```
##              Procedure Process_Depts;
##              const
                        MIN_TOT_SALES = 50000.00;
##              type
                        {Dept_Rec corresponds to the "dept" table}
##                       Dept_Rec = record
##                              name: String12;
##                              totsales: Long_Float;
##                              employees: Short_Integer;
##                      end;
##              var
##                      no_rows:      Integer;
##                      emps_term:    Integer; {Employees terminated}
                        deleted_dept: Boolean; {Was the dept deleted?}
                        dept_format:  String20; {Formatting value}
##                      dpt:          Dept_Rec;

##              begin {Process_Depts}

##                      range of d is dept
##                      declare cursor deptcsr for
##                              retrieve (d.name, d.totsales, d.employees)
##                              for direct update of (name, employees)

                        no_rows := 0;
                        emps_term := 0;
##                      open cursor deptcsr

                        while (no_rows = 0) do begin
##                               retrieve cursor deptcsr (dpt.name,
##                                                        dpt.totsales,
##                                                        dpt.employees)
##                      inquire_equel (no_rows = endquery)
                        if (no_rows = 0) then begin

                                {Did the department reach minimum sales?}
                                if (dpt.totsales < MIN_TOT_SALES) then begin
##                                      delete cursor deptcsr
                                        deleted_dept := TRUE;
                                        dept_format := ' -- DISSOLVED --';
                                end else begin
                                        deleted_dept := False;
                                        dept_format := ' ';
                                end; {If reached minimum sales}
                                {Log what we have just done}
                                 Write('Department: ', dpt.name);
                                 Write(', Total Sales: ', dpt.totsales:12:3);
                                 Writeln(dept_format);

                        {Now process each employee in the department}
                        Process_Employees(dpt.name, deleted_dept, emps_term);

                        {If employees were terminated, record it}
                        if ((emps_term > 0) and (not deleted_dept)) then
##                               replace cursor deptcsr
##                               (employees = employees - emps_term)

                        end; {If a row was retrieved}

                         end; {Continue with cursor loop}
##                      close cursor deptcsr

##              end; {Process_Depts}
```

```
##              begin {Process_Expenses}
                       Writeln('Entering application to process expenses.');
                       if (Init_Db) then begin
                                Process_Depts;
                                End_Db;
                                Writeln('Completion of application.');
                       end;
##              end; {Process_Expenses}

##              begin {main}
                       Process_Expenses;
                       Exit_Program:;
##              end. {main}
```

The Employee Query Interactive Forms Application

This EQUEL/FORMS application uses a form in **query** mode to view a subset of the Employee table in the Personnel database. An Ingres query qualification is built at runtime using values entered in fields of the form "empform."

The objects used in this application are:

Object	Description
personnel	The program's database environment.
employee	A table in the database, with six columns: name (**c20**) age (**i1**) idno (**i4**) hired (**date**) dept (**c10**) salary (**money**).
empform	A VIFRED form with fields corresponding in name and type to the columns in the Employee database table. The Name and Idno fields are used to build the query and are the only updatable fields. "Empform" is a compiled form.

The application is driven by a **display** statement that allows the runtime user to enter values in the two fields that will build the query. The Build_Query and Exec_Query procedures make up the core of the query that is run as a result. Note the way the values of the query operators determine the logic used to build the **where** clause in Build_Query. The **retrieve** statement encloses a **submenu** block that allows the user to step through the results of the query.

No updates are performed on the retrieved values, but any particular employee screen may be saved in a log file through the **printscreen** statement.

The following **create** statement describes the format of the Employee database table:

```
##   create employee
##          (name         = c20,  { Employee name }
##           age        = i1,     { Employee age }
##           idno       = i4,     { Unique employee id }
##           hired      = date,   { Date of hire }
##           dept       = c10,    { Employee department }
##           salary     = money)  { Annual salary }

## program Employees;
## type
##      String2 = packed array[1..2] of Char;
##      String10 = packed array[1..10] of Char;
##      String20 = packed array[1..20] of Char;
##      String25 = packed array[1..25] of Char;
##      VString100 = varying[100] of Char;
##      Float = Real;
##      Short_Integer = [Word] -32768 .. 32767;
## var
##      empform : [External] Integer;
## declare

## procedure Employee_Query;
## var
        {Global WHERE clause qualification buffer}
##      where_clause: VString100;

        {
        | Procedure: Build_Query
        | Purpose: Build an Ingres query from the values in the
        |       "name" and "idno" fields in "empform".
        | Parameters:
        |       None
        }
##      procedure Build_Query;
##      type
##          opers = array[1..6] of String2;
##      var
##           ename: String20;
##           eidno: Integer;

            {
            | Query operator table that maps integer values to
            | string query operators.
            }
##          operators: opers;

            {
            | Operators corresponding to the two fields,
            | that index into the "operators" table.
            }
##          name_op, id_op: Integer;

##      begin {Build_Query}
            operators := opers ('= ', '!=', '< ', '> ', '<=', '>=');
##          getform #empform
##              (ename = name, name_op = getoper(name),
##               eidno = idno, id_op = getoper(idno))

            {Fill in the WHERE clause}
            if ((name_op = 0) and (id_op = 0)) then
            begin
                where_clause := '1=1'; {Default qualification}
            end else if ((name_op = 0) and (id_op <> 0)) then
```

```
            begin
                {Query on the "idno" field}
                WriteV( where_clause,
                         'e.idno', operators[id_op],
                         eidno);
            end else if ((name_op <> 0) and (id_op = 0)) then
            begin
                {Query on the "name" field}
                 where_clause :=
                         'e.name' + operators[name_op] +
                        '"' + ename + '"';
            end else { ((name_op <> 0) and (id_op <> 0)) }
            begin
                {Query on both fields}
                WriteV( where_clause,
                         'e.name', operators[name_op],
                         '"', ename, '" and ',
                         'e.idno', operators[id_op],
                         eidno);
             end;
##       end; {Build_Query}

        {
        | Procedure: Exec_Query
        | Purpose:   Given a query buffer, defining a WHERE clause
        |            issue a RETRIEVE to allow the runtime use to
        |            browse the employees found with the given qualification.
        | Parameters:
        |            None
        }

##      procedure Exec_Query;
##      var
##           ename:     String20; {Employee data}
##           eage:      Short_Integer;
##           eidno:     Integer;
##           ehired:    String25;
##           edept:     String10;
##           esalary:   Float;
##           rows:      Integer; {Were rows found}
##      begin {Exec_Query}
                {Issue query using WHERE clause}
##              retrieve (
##                      ename = e.name, eage = e.age,
##                      eidno = e.idno, ehired = e.hired,
##                      edept = e.dept, esalary = e.salary)
##              where where_clause
##              begin {retrieve}
                        {Put values up and display them}
##                      putform #empform (
##                              name = ename, age = eage,
##                              idno = eidno, hired = ehired,
##                              dept = edept, salary = esalary)
##                      redisplay

##                      submenu
##                      activate menuitem 'Next', frskey4
##                      begin
                                {
                                | Do nothing, and continue with the
                                | retrieve loop. The last one will
                                | drop out.
                                }
##                      end {Next}

##                      activate menuitem 'Save', frskey8
```

```
##                      begin
                                {Save screen data in log file}
##                              printscreen (file = 'query.log')
                                {Drop through to next employee}
##                      end {Save}

##                      activate menuitem 'End', frskey3
##                      begin
                                {Terminate the RETRIEVE loop}
##                              endretrieve
##                      end {End}
##              end {retrieve}
##              inquire_equel (rows = rowcount)
                if (rows = 0) then
                begin
##                      message 'No rows found for this query'
                end else
                begin
##                      clear field all
##                      message 'No more rows. Reset for next query'
                end;
##              sleep 2
##          end; {Exec_Query}

##      begin {Employee_Query}
##          forms
##          message 'Accessing Employee Query Application . . .'
##          ingres personnel

##          range of e is employee

##          addform empform

##          display #empform query
##          initialize

##          activate menuitem 'Reset'
##          begin
##              clear field all
##          end {Reset}

##          activate menuitem 'Query'
##          begin
                {Verify validity of data}
##               validate
                Build_Query;
                Exec_Query;
##           end {Query}

##          activate menuitem 'LastQuery'
##          begin
                Exec_Query;
##          end {LastQuery}

##          activate menuitem 'End'
##          begin
##               breakdisplay
##              end {End}
##          finalize

##          clear screen
##          endforms
##          exit
##      end; {Employee_Query};

##      begin {main}
```

```
                Employee_Query;
##      end. {main}
```

The Table Editor Table Field Application

This EQUEL/FORMS application uses a table field to edit the Person table in the Personnel database. It allows the user to update a person's values, remove the person, or add new persons. Various table field utilities are provided with the application to demonstrate their use and their interaction with an Ingres database.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
person	A table in the database, with three columns: name (**c20**) age (**i2**) number (**i4**). Number is unique.
personfrm	The VIFRED form with a single table field.
persontbl	A table field in the form, with two columns: name (**c20**) age (**i4**) When initialized, the table field includes the hidden number (**i4**) column.

At the start of the application, a **retrieve** statement is issued to load the table field with data from the Person table. Once the table field has been loaded, the user can browse and edit the displayed values. Entries can be added, updated or deleted. When finished, the values are unloaded from the table field, and, in a multi-statement transaction, the user's updates are transferred back into the Person table.

The following **create** statement describes the format of the Person database table:

```
##  create person
##      (name   = c20,  { Person name }
##       age    = i2,   { Age }
##       number = i4)   { Unique id number }

##  program TableEdit( input, output );
##  type
##       String13 = packed array[1..13] of Char;
##       String20 = packed array[1..20] of Char;
##       String80 = packed array[1..80] of Char;
```

```
##       Short_Integer = [Word] -32768 .. 32767;
##  declare

## procedure Table_Edit;
    label
        exit_label;
## type
##      {Table field row states}
##      RowStates = (
##      RowUndef,        {Empty or undefined row}
##      RowNew,          {Appended by user}
##      RowUnchange,     {Loaded by program - not updated}
##      RowChange,       {Loaded by program and updated}
##      RowDelete        {Deleted by program}
##      );

## var
##      {Person information corresponds to "person" table}
##      pname:  String20;        {Full name}
##      page:   Short_Integer;   {Age}
##      pnumber:Integer;         {Unique person number}
##      pmaxid: Integer;         {Maximum person id number}

##      {Table field entry information}
##      state:  RowStates;       {State of data set row (see above)}
##      recnum,                  {Record number}
##      lastrow: Integer;        {Lastrow in table field}

        {Utility buffers}
##       search:         String20;       {Name to find in search loop}
##       msgbuf:         String80;       {Message buffer}
##       password:       String13;       {Password buffer}
##       respbuf:        Char;           {Response buffer}

        {Error handling variables for database updates}
##      upd_err,                 {Updates error}
##      upd_rows:     Integer;   {Number of rows updated}
        upd_commit:   Boolean;   {Commit updates}
##      save_changes: Boolean;   {Save changes or Quit}

##  begin {Table_Edit}

        {
        | Start up Ingres and the Ingres/Forms system
        | We assume no Ingres errors will happen during
        | screen updating
        }

##      ingres personnel

##      forms

        {Verify that the user can edit the "person" table}
##      prompt noecho ('Password for table editor: ', password)
        if (password <> 'MASTER_OF_ALL') then
        begin
##          message 'No permission for task. Exiting . . .'
##          endforms
##          exit
            goto exit_label;
        end;

##      message 'Initializing Person Form . . .'
##      forminit personfrm

        {
```

```
        | Initialize "persontbl" table field with a data set
        | in FILL mode so that the runtime user can append
        | rows. To keep track of events occurring to original
        | rows that will be loaded into the table field, hide
        | the unique person number.
        }
##      inittable personfrm persontbl fill (number = integer)
        {
        | Load the information from the "person" table into the
        | person variables. Also save away the maximum person
        | id number.
        }
##      message 'Loading Person Information . . .'

##      range of p IS person

        {Fetch data into person record, and load table field}
##      retrieve (pname = p.name, page = p.age,
##                pnumber = p.number)
##      begin
##              loadtable personfrm persontbl
##                  (name = pname, age = page, number = pnumber)
##      end {Retrieve}

        {
        | Fetch the maximum person id number for later use.
        | Performance Note: max will do sequential scan of table.
        }
##      retrieve (pmaxid = max(p.number))

        {Display the form and allow runtime editing}

##      display personfrm update
##      initialize

##      {
##      | Provide a menu, as well as the system FRS key to scroll
##      | to both extremes of the table field. Note that a comment
##      | between DISPLAY loop components MUST be marked with a
##.     }
##

##      activate menuitem 'Top', frskey5
##      begin
##      scroll personfrm persontbl TO 1 {Backward}
##      end {Top}

##      activate menuitem 'Bottom', frskey6
##      begin
##              scroll personfrm persontbl to end{Forward}
##      end {Bottom}

##      activate menuitem 'Remove'
##      begin
                {
                | Remove the person in the row the user's cursor
                | is on. If there are no persons, exit operation
                | with message. Note that this check cannot
                | really happen as there is always at least one
                | UNDEFINED row in FILL mode.
                }

##      inquire_frs table personfrm
##                      (lastrow = lastrow(persontbl))
        if (lastrow = 0) then
        begin
```

```
##          message 'Nobody to Remove'
##          sleep 2
##          resume field persontbl
        end;

##      deleterow personfrm persontbl {Recorded for later}
##     end {Remove}

##      activate menuitem 'Find', frskey7
##      begin
                {
                | Scroll user to the requested table field entry.
                | Prompt the user for a name, and if one is typed
                | in loop through the data set searching for it.
                }

                search := ' ';
##              prompt ('Person''s name : ', search)
                if (search[1] = ' ') then
##                      resume field persontbl

##              unloadtable personfrm persontbl
##                      (pname = name, recnum = _record,
##                       state = _state)
##          begin
                        {Do not compare with deleted rows}
                        if ((state <> RowDelete) and (pname = search))
                                then
                        begin
##                              scroll personfrm persontbl to recnum
##                              resume field persontbl
                        end;
##          end; {Unloadtable}

            {Fell out of loop without finding name. Issue error.}
            msgbuf := 'Person ''' + search +
                        ''' not found in table. [HIT RETURN] ';
##          prompt noecho (msgbuf, respbuf)
##      end {Find}

##      activate menuitem 'Save', frskey8
##      begin
##              validate field persontbl
                save_changes := TRUE;
##              breakdisplay
##      end {Save}

##      activate menuitem 'Quit', frskey2
##      begin
                save_changes := FALSE;
##              breakdisplay
##      end             {Quit}
##      finalize

        if (not save_changes) then {Quit application}
        begin
##              endforms
##              exit
                goto exit_label;
        end;

        {
        | Exit person table editor and unload the table field.
        | If any updates, deletions or additions were made,
        | duplicate these changes in the source table. If the
        | user added new people we must assign a unique person
```

```
        | id before returning it to the database table. To do
        | this, we increment the previously saved maximum id
        | number with each APPEND.
        }

##      message 'Exiting Person Application . . .'

        {
        | Do all the updates in a multi-statement transaction
        | (for simplicity, this transaction does not restart on
        | deadlock error).
        }
##      begin transaction
        upd_commit := TRUE;

        {
        | Handle errors in the UNLOADTABLE loop, as we want to
        | cleanly exit the loop, after cleaning up the transaction.
        }

##      unloadtable personfrm persontbl
##              (pname = name, page = age,
##               pnumber = number, state = _state)
##      begin
                case (state) of
                RowNew:
                        begin
                                {
                                | Filled by user.
                                | Insert with new unique id
                                }
                                pmaxid := pmaxid + 1;
##                              repeat append to person
##                                      (name = @pname,
##                                       age = @page,
##                                       number = @pmaxid);
                        end; {RowNew}

                RowChange:
                        begin
                                {Updated by user. Reflect in table}
##                              repeat replace p
##                                      (name = @pname, age = @page)
##                                      where p.number = @pnumber
                        end; {RowChange}

                RowDelete:
                        begin
                                {
                                | Deleted by user, so delete from table.
                                | Note that only original rows are saved
                                | by the program, and not rows appended
                                | at runtime.
                                }
##                              repeat delete p
##                                      where p.number = @pnumber
                        end; {RowDelete}

                otherwise
                        begin
                                {
                                | Else UNDEFINED or UNCHANGED
                                | No updates required.
                                }
                                ;
                    end; {Otherwise}
```

```
                end; {case}

                {
                | Handle error conditions -
                | If an error occurred, then abort the transaction.
                | If a no rows were updated then inform user, and
                | prompt for continuation.
                }

##              inquire_equel (upd_err = errorno, upd_rows = rowcount)

                if (upd_err> 0) then {Abort on error}
                begin
                        upd_commit := FALSE;
##                      message 'Aborting updates . . .'
##                      abort
##                      endloop
                end else if (upd_rows = 0) then {May want to stop}
                begin
                        msgbuf := 'Person ''' + pname +
                                        ''' not updated. Abort all updates? ';
##                      prompt noecho (msgbuf, respbuf)
                        if ((respbuf = 'Y') or (respbuf = 'y')) then
                        begin
                                upd_commit := FALSE;
##                              abort
##                              endloop
                        end;
                end;
##          end; {unloadtable}

            if (upd_commit) then
##              end transaction {Commit the updates}

##          endforms {Terminate the Forms and Ingres}
##          exit

        exit_label:
##      end; {Table_Edit}

##      begin {main}
            Table_Edit;
##      end. {main}
```

The Professor-Student Mixed Form Application

This EQUEL/FORMS application lets the user browse and update information about graduate students who report to a specific professor. The program is structured in a master/detail fashion, with the professor being the master entry, and the students the detail entries. The application uses two forms—one to contain general professor information and another for detailed student information.

The objects used in this application are:

Object	Description
personnel	The program's database environment.
professor	A database table with two columns: pname (**c(25)**) pdept (**c(10)**) See its **create** statement below for a full description.
student	A database table with seven columns: sname (**c(25)**) sage (**i1**) sbdate (**c(25)**) sgpa (**f4**) sidno (**i1**) scomment (**text(200)**) sadvisor (**c(25)**) See the **create** statement below for a full description. The sadvisor column is the join field with the pname column in the Professor table.
masterfrm	The main form has the pname and pdept fields, which correspond to the information in the Professor table, and the studenttbl table field. The pdept field is display-only. "Masterfrm" is a compiled form.
studenttbl	A table field in "masterfrm" with the sname and sage columns. When initialized, it also has five hidden columns corresponding to information in the Student table.
studentfrm	The detail form, with seven fields, which correspond to information in the Student table. Only the fields sgpa, scomment, and sadvisor are updatable. All other fields are display-only. "Studentfrm" is a compiled form.

Object	Description
grad	A global structure, whose members correspond in name and type to the columns of the Student database table, the "studentfrm" form and the studenttbl table field.

The program uses the "masterfrm" as the general-level master entry, in which data can only be retrieved and browsed, and the "studentfrm" as the detailed screen, in which specific student information can be updated.

The runtime user enters a name in the pname (professor name) field and then selects the **Students** menu operation. The operation fills the displayed and hidden columns or table field "studenttbl" with detailed information of the students reporting to the named professor. The user can then browse the table field (in **read** mode), which displays only the names and ages of the students. More information about a specific student can be requested by selecting the **Zoom** menu operation. This operation displays the form "studentfrm." The fields of "studentfrm" are filled with values stored in the hidden columns of "studenttbl." The user can make changes to three fields (sgpa, scomment, and sadvisor). If validated, these changes will be written back to the database table (based on the unique student id), and to the table field's data set. This process can be repeated for different professor names.

```
##   create student  { Graduate Student table }
##      (sname          = c25,          { Name }
##       sage           = i1,           { Age }
##       sbdate         = c25,          { Birth date }
##       sgpa           = f4,           { Grade point average}
##       sidno          = i4,           { Unique student number }
##       scomment       = text(200),    { General comments }
##       sadvisor       = c25)          { Advisor's name}

##  create professor    {Professor table}
##      (pname  = c25,  {Professor's name}
##       pdept  = c10)  {Department}

##  program University;

##  const
##
##          shortstrlen = 10;
##          mediumstrlen = 25;
##          longstrlen = 100;
    type
##          StrShort    = packed array [1..SHORTSTRLEN] of Char;
##          StrMedium   = packed array [1..MEDIUMSTRLEN] of Char;
##          StrLong     = packed array [1..LONGSTRLEN] of Char;
##          NatTiny     = [byte] 0..255; {A one-byte unsigned integer}
##  var
##          {Master and student compiled forms}
##          masterfrm,studentfrm: [external] Integer;

##  declare

##  {
##  | Procedure: Prof_Student
##  | Purpose:  Main body of "Professor/Student"
##  | Master-Detail Application.
##  }
```

```
##  procedure Prof_Student;

##  type
##          {Grad student record maps to "student" DB table}
##          Student_Rec = record
##              sname: StrMedium;
##              sage: NatTiny;
##              sbdate: StrMedium;
##              sgpa: Real;
##              sidno: Integer;
##              scomment: StrMedium;
##              sadvisor: StrMedium;
##          end;

##          {Professor record maps to "professor" DB table}
##          Prof_Rec = record
##              pname: StrMedium;
##              pdept: StrShort;
##          end;

##      var
##          grad: Student_Rec;
##          prof: Prof_Rec;
##          old_advisor: StrMedium;      {Advisor before ZOOM}

##          {Useful forms runtime information}
##          lastrow,             {Lastrow in table field}
##          istable: Integer;    {Is a table field?}

##          {Utility buffers}
##          msgbuf: StrLong:     {Message buffer}
##          respbuf: Char;       {Response buffer}
##          {
##          | Function: Student_Infor_Changed
##          | Purpose: Allow the user to zoom into the details
##          | of a selected student. Some of the data can be
##          | updated by the user. If any updates were made,
##          | then reflect these back into the database table.
##          | The procedure returns TRUE if any changes were made.
##          | Parameters:
##          | None.
##          | Returns:
##          | TRUE/FALSE - Changes were made to the database.
##          | Sets the global "grad" record with the new data.
##          }

##          function Student_Info_Changed : Boolean;

##          var
##              changed: Integer; {Changes made to the form?}
##              valid_advisor: Integer; {Is the advisor a professor?}

##          begin {Student_Info_Changed}

##                      {Display the detailed student information}
##                      display #studentfrm update

##                      initialize (sname = grad.sname,
##                                  sage  = grad.sage,
##                                  sbdate = grad.sbate,
##                                  sgpa    = grad.sgpa,
##                                  sidno   = grad.sidno,
##                                  scomment = grad.scomment,
##                                  sadvisor = grad.sadvisor)
```

```
##                      activate menuitem 'Write'
##                      begin
##                              {
##                              | If changes were made then update the
##                              | database table. Only bother with the
##                              | fields that are not read-only.
##                              }
##                              inquire_frs form (changed = change)

                                if (changed = 1) then
                                begin
##                                      validate
##                                      getform (grad.sgpa = sgpa,
##                                               grad.scomment = scomment,
##                                              grad.sadvisor = sadvisor)

##                                      {
##                                      | Enforce referential integrity.
##                                      | If there aren't any professors
##                                      | matching the advisor's name then
##                                      | don't change it -- user would never
##                                      | be able to access it again to fix it.
##                                      }

##                      retrieve (valid_advisor =
##                              count(p.pname
##                                 where p.pname =
##                                    grad.sadvisor))

                        if (valid_advisor <= 0) then
                        begin
##                        message 'Not a valid professor'
##                        sleep 2
##                        resume field sadvisor
                        end else
                        begin
##                              message 'Writing to database...'
##                              replace s(sgpa = grad.sgpa,
##                                      scomment = grad.scomment,
##                                      sadvisor = grad.sadvisor)
##                              where s.sidno = grad.sidno)
##                              breakdisplay
                        end; {Valid advisor}
                    end; {Form was changed}
##          end {write}

##          activate menuitem 'End', frskey3
##          begin
                {Quit without submitting changes}
                 changed :=0;
##               breakdisplay
##          end {End}

##          finalize
            Student_Info_Changed := (changed =1);

##          end; {Student_Info_Changed}

##      begin {Prof_Student}

##          {Start up Ingres and the Forms system}
##          forms
##          message 'Initializing Student Administrator...'

##          ingres personnel
```

```
##          range of p IS professor, s is student

##          addform masterfrm
##          addform studentfrm

##          {
##          | Initialize "studenttbl" with a data set in READ mode.
##          | Declare hidden columns for all the extra fields that
##          | the program will display when more information is
##          | requested about a student. Columns "sname" and "sage"
##          | are displayed, all other columns are hidden, to be
##          | used in the student information form.
##          }

##          inittable #masterfrm studenttbl read
##              (sbdate  = char(25),
##              sgpa     = float4,
##              sidno    = integer4,
##              scomment = char(200),
##              sadvisor = char(20))

##          {
##          | Drive the application, by running "masterfrm", and
##          | allowing the user to "zoom" into a selected student.
##          }
##          display #masterfrm update

##          initialize
##          begin
##              message 'Enter an Advisor name...'
##              sleep 2
##          end {Initialize}

##          activate menutiem 'Students',field 'pname'
##          begin
##              {Load the students of the specified professor}
##              getform (prof.pname = pname)

##              {If no professor name is given then resume}
                if (prof.pname[1] = ' ') then
##                      resume field pname

##          {
##          | Verify that the professor exists. If not
##          | print a message, and continue. We assume that
##          | each professor has exactly one department.
##          }
            prof.pdept :=' ';
##          retrieve (prof.pdept = p.pdept)
##          where p.pname = prof.pname

##          {If no professor report error}
            if (prof.pdetp[1] = ' ') then
            begin
                msgbuf := 'No professor with name ''+
                            prof.pname + '' [RETURN]';
##              prompt noecho (msgbuf, respbuf)
##              clear field all
##              resume field pname
            end;

##          {Fill the department field and load students}
##          message 'Retrieving Student Information...'

##          putform (pdept = prof.pdept)
##          clear field studenttbl
```

```
##          redisplay   {Refresh for query}

##          {
##          | With the advisor name, load into the "studenttbl'
##          | table field all the graduate students who report
##          | to the professor with that name.
##          | Columns "sname" and "sage" will be displayed, and
##          | all other columns will be hidden.
##          }
##          retrieve (grad.sname = s.sname,
##                      grad.sage = s.sage,
##                      grad.sbdate = s.sbdate,
##                      grad.sgpa = s.sgpa,
##                      grad.sidno = s.sidno,
##                      grad.scomment = s.scomment,
##                      grad.sadvisor = s.sadvisor)
##                      where s.advisor = prof.pname
##          begin
##                      loadtable #masterfrm studenttbl
##                      (sname = grad.sname,
##                       sage = grad.sage,
##                       sbdate = grad.sbdate,
##                       sgpa = grad.sbdate,
##                       sidno = grad.sidno,
##                       scomment = grad.scomment,
##                       sadvisor = grad.sadvisor)
##              end {Retrieve}
##              resume field studenttbl
##          end {Students}

##          activate menuitem 'Zoom'
##          begin
##               {
##              | Confirm that user is on "studenttbl", and that
##              | the table field is not empty. Collect data from
##              | the row and zoom for browsing and updating.
##              }
##              inquire_frs field #masterfrm (istable =table)
                if (istable = 0) then
                begin
##                      prompt noecho
##                              ('Select from the student table [RETURN]',
##                               respbuf)
##                      resume field studenttbl
                end;

##               inquire_frs table #masterfrm (lastrow = lastrow)
                 if (lastrow = 0) then
                 begin
##                      prompt noecho
##                              ('There are no students [RETURN]',
##                               respbuf)
##                      resume field pname
                 end;

##              {Collect all data on student into graduate record}
##              getrow #masterfrm studenttbl
##                      (grad.sname = sname,
##                       grad.sage = sage,
##                       grad.sbdate = sbdate,
##                       grad.sgpa = sgpa,
##                       grad.sidno = sidno,
##                       grad.scomment = scomment,
##                       grad.sadvisor = sadvisor)

##              {
```

```
##              | Display "studentfrm", and if any changes were made
##              | make the updates to the local table field row.
##              | Only make updates to the columns corresponding to
##              | writable fields in "studentfrm".
##              | If the student changed advisors then delete this row
##              | from the current display -- it no longer belongs here.
##              }
                old_advisor := grad.sadvisor;
                if (Student_Infor_Changed) then
                begin
                        if (grad.sadvisor <> old_advisor) then
                        begin
##                              deleterow #masterfrm studenttbl
                        end else
                        begin
##                              putrow #masterfrm studenttbl
##                                      (sgpa = grad.sgpa,
##                                       scomment = grad;scomment,
##                                       sadvisor = grad.sadvisor)
                        end;
                end; {If student info changed}
##          end {Zoom}

##          activate menuitem 'QUIT', frskey2
##          begin
##               breakdisplay
##          end {Quit}

##          finalize

##          clear screen
##          endforms
##          exit
##  end; {Prof_Student}

##  begin {University}
            Prof_Student;
##  end. {University}
```

Index

#

.

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

W